INFORMATIQUE & COMMUNICATION

COLLECTION A. BIANCIOTTO - P. BOYE

L'ALGORITHMIQUE ET LE PASCAL

DANIEL CANEVET

DELAGRAVE

INFORMATIQUE ET COMMUNICATION
Sous la direction de A.Bianciotto et P.Boye

L'ALGORITHMIQUE ET LE PASCAL

Daniel CANEVET
Responsable Informatique
Caisse Maladie Régionale des Pays de la Loire
Enseignant en Informatique
Faculté des Sciences et des Techniques de Nantes

A L'USAGE :

- des étudiants en DEUG Scientifiques, BTS, DUT Informatique
- des élèves-professeurs des Instituts Universitaires de Formation des Maîtres
- des élèves des classes préparatoires
- des élèves des classes à option informatique
- des élèves des classes des baccalauréats technologiques
- des auditeurs de la formation continue

DELAGRAVE
15, rue Soufflot, 75240 PARIS Cédex 05

DANS LA COLLECTION TECHNIQUES ET NORMALISATION

A. BIANCIOTTO – P. BOYE
L'INFORMATIQUE EN AUTOMATISATION INDUSTRIELLE

DANS LA COLLECTION ESPACES TECHNOLOGIQUES

C. ROBINET – A. BIANCIOTTO – P. BOYE
LA LOGIQUE ELECTRONIQUE

P. SALETTE – A. BIANCIOTTO – P. BOYE
L'ELECTRONIQUE ET SES FONCTIONS (2 TOMES)

A. RIDEAU – A. BIANCIOTTO – P. BOYE
LA TECHNOLOGIE DES SYSTEMES AUTOMATISES (Seconde TSA)

La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective" et d'autre part, que les analyses et courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite" (alinéa 1er de l'article 40).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal.

© Delagrave, 1992
ISBN : 2-206-00698-7

AVANT-PROPOS

Cet ouvrage propose un apprentissage simultané de l'**Algorithmique**, outil de modélisation pour la résolution de problèmes, et d'un langage de programmation évolué : le **langage Pascal**, dont les structures sont calquées sur celles de l'Algorithmique.

L'algorithmique, utilisant une syntaxe simple, proche du langage naturel, permet à l'analyste d'un problème de se concentrer non pas sur la forme, mais sur le fond, c'est à dire sur la **méthode de résolution**, en structurant la solution.

L'auteur du langage Pascal énonçait, dès l'origine, son objectif fondamental : Le Pascal devait être "**adapté à l'enseignement de la programmation rigoureuse et structurée**". Les qualités du Pascal, allant bien au delà de l'objectif initial de l'auteur, font que ce langage est de plus en plus utilisé.

Devant les succès du Pascal, des extensions ont été rajoutées, introduisant ainsi de nouvelles versions pour en augmenter la puissance et l'efficacité.
La version de Pascal choisie pour cet ouvrage est définie par la norme **NF Z 65-300**, correspondant à la norme internationale **ISO 7185**.
Le type chaîne de caractères n'est pas connu de ces versions de base ; il est cependant si répandu dans les nouvelles versions, qui le définissent comme type STRING, que nous l'évoquerons dans cet ouvrage.

Chaque point étudié est présenté selon un même modèle :

- **un exposé** des définitions et syntaxes algorithmique et Pascal,
- **de brefs exemples** sur ces définitions,
- **un exemple numérique simple** d'algorithme et de programme,
- **des remarques** sur les particularités,
- **des applications** commentées et des énoncés d'exercices.

Plus de 50 Algorithmes et de 50 programmes correspondants permettent un apprentissage par l'exemple, respectant une complexité croissante.

- Les mots clés de l'Algorithme sont soulignés.
- Les mots réservés du Pascal sont en MAJUSCULE.
- Les noms de variables citées dans le texte sont en *italique*.
- Les encadrés grisés sont réservés aux programmes Pascal.
- Les algorithmes et programmes correspondants font apparaître en **caractères gras** les instructions étudiées et les points importants.

L'Auteur

TABLE DES MATIERES

page

A – INTRODUCTION AUX LANGAGES EVOLUES

I – COMMUNICATION HOMME MACHINE 5
- L'Ordinateur – Le Langage Binaire
- Langage d'assemblage et assembleur
- Langages évolués

II – ALGORITHMIQUE 8
- Définition – Organigramme
- Langage d'expression algorithmique
- Variables et Trace de l'Algorithme

III – CHOIX DU LANGAGE PASCAL 11
- Raisons du Choix
- Structure du Langage
- Normalisation du Pascal

B – ALGORITHMIQUE ET PASCAL

I – BASES DES LANGAGES
1. Caractéristiques 13
2. Instructions simples 20
3. Expressions 23

II – INSTRUCTIONS STRUCTUREES
4. Instruction composée 29
5. Instructions Alternatives 29
6. Instructions de conditions multiples 35

III – INSTRUCTIONS REPETITIVES
7. Traitement d'une File 38
8. Structure Pour...Fin Pour 38
9. Structure Tant que...Fin Tant que 42
10. Structure Répéter...Jusqu'à 45

IV – PROCEDURES ET FONCTIONS
11. Analyse Descendante 49
12. Procédures 51
13. Fonctions 61
14. Récursivité 65

V – STRUCTURES DE DONNEES
15. Vecteur ou Tableau à une dimension 66
16. Tableaux à plusieurs dimensions 76
17. Ensembles 79
18. Structure d'Enregistrement 82
19. Pointeurs 86
20. Fichiers 90

VI – ANNEXES
A. Mots réservés – Identificateurs prédéfinis 94
B. Index 95

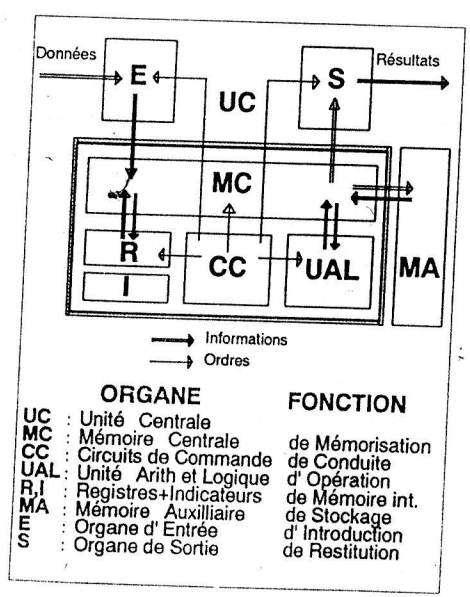
A – INTRODUCTION AUX LANGAGES EVOLUES

I – COMMUNICATION HOMME MACHINE

1. L'ORDINATEUR

L'ordinateur est aujourd'hui un outil très puissant qui assiste l'Homme dans de nombreux domaines d'activité. Il contribue largement aux progrès des Sciences et des Techniques.
L'évolution de la communication entre l'Homme et la machine correspond aux différentes formes de langages que l'Homme a créé pour dialoguer avec l'ordinateur.

La figure A–1. fournit une présentation schématique de cet outil, permettant de repérer ses composantes fondamentales.



ORGANE		FONCTION
UC	: Unité Centrale	
MC	: Mémoire Centrale	de Mémorisation
CC	: Circuits de Commande	de Conduite
UAL	: Unité Arith et Logique	d' Opération
R,I	: Registres+Indicateurs	de Mémoire int.
MA	: Mémoire Auxilliaire	de Stockage
E	: Organe d' Entrée	d' Introduction
S	: Organe de Sortie	de Restitution

Fig. A–1. Schéma fonctionnel de l'ordinateur

– La **Mémoire Centrale** sert exclusivement à stocker le programme et les données nécessaires au traitement.

– Les **Circuits de Commande** décodent le programme inscrit en mémoire centrale et ordonnent l'exécution des instructions.

– L'**Unité Arithmétique et Logique** effectue les calculs sur les données.

– Des **liaisons câblées** font circuler les informations entre les différentes parties. Elles portent le nom de **BUS**.

2. LANGAGE BINAIRE

L'ordinateur ne connaît que le langage binaire dont l'alphabet est représenté par 2 caractères (0 et 1).

Lors de l'exécution d'un programme, la mémoire, les registres, les tampons ... prennent des états binaires stables, affectés par la logique câblée d'un ensemble de micro–instructions. Le fonctionnement de l'ordinateur suit les lois de l'Algèbre de Boole : Ensemble des règles s'appliquant aux systèmes binaires.

Pour se faire comprendre de l'ordinateur , les premiers programmeurs devaient lui adresser des séries d'instructions composées de 0 et de 1. Ces instructions, adaptées au micro–processeur, nécessitaient une connaissance approfondie du système (Structure des instructions, Lexique des codes, Modes d'adressage, Mémoire, Organes d'entrée–sortie, ...).

La programmation dans ces conditions était lente, source d'erreur, et s'avérait ardue et fastidieuse (Fig. A-2).

Le jeu d'instructions peut varier d'une machine à l'autre, mais on retrouve généralement les types suivants :

- **Instructions de transferts** entre mémoire centrale et/ou registres du micro-processeur,
- **Instructions de traitement, opérations arithmétiques et logiques,**
- **Instructions d'enchaînement, sauts et branchements,**
- **Instructions d'entrée-sortie,**
- **Instructions de commande et de test d'état.**

L'instruction, enregistrée en mémoire centrale, contient :

- **Un Code Opération**, caractérisant la fonction à effectuer,
- **Une ou plusieurs Adresses** (opérandes, résultat).

L'adresse de l'instruction suivante peut être, soit contenue dans l'instruction en cours, soit contenue dans un registre spécialisé : le **compteur ordinal**, qui calcule l'adresse de l'instruction suivante en ajoutant la longueur de l'instruction en cours.

Considérons par exemple l'instruction binaire suivante qui charge dans le registre accumulateur la valeur contenue en mémoire à l'adresse 1000 :

1010 1101 1110 1000 0000 0011.

Les 8 premiers bits fournissent un code opérateur et les 16 bits suivants indiquent l'adresse sur laquelle porte l'opérateur.

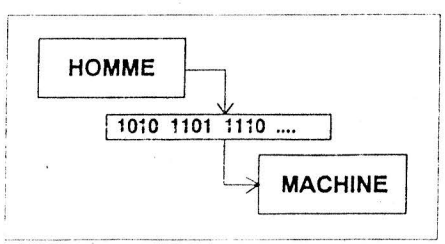


Fig. A-2. Langage binaire

3. LANGAGE D'ASSEMBLAGE ET ASSEMBLEUR

Le langage d'assemblage est un langage proche du langage machine, donnant plus de facilité dans la rédaction des programmes. Il reste néanmoins **intimement lié au micro-processeur** : un programme établi pour un type de machine n'est pas exécutable sur une machine basée sur un micro-processeur différent.

Le langage d'assemblage, abusivement appelé Assembleur, met en oeuvre un système de numération, non plus binaire, mais hexadécimal.

L'alphabet du langage se compose des chiffres de 0 à 9, et des lettres de A (équivalent du décimal 10) à F (équivalent du décimal 15).

Revenons à notre code binaire de l'exemple précédent : 1010 1101 1110 1000 0000 0011
La conversion en hexadécimal de ce code donne :

1010 = **A**, 1101 = **D**, 1110 = **E**,
1000 = **8**, 0000 = **0**, 0011 = **3**
soit **AD E8 03.**

Le code opération est AD, que l'on peut définir comme le code mnémonique LDA (LoaD Accumulateur). L'adresse est E8 03.

L'ASSEMBLEUR est le programme permettant de passer du langage d'assemblage au binaire c'est à dire, de traduire LDA E8 03 en 1010 1101 1110 1000 0000 0011 **(Fig. A-3).**

Le programme SOURCE écrit en langage d'assemblage, est traduit par l'ASSEMBLEUR, qui va fournir le programme OBJET écrit en binaire.

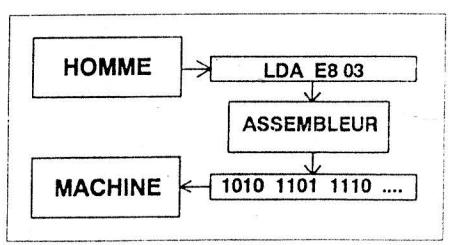


Fig. A-3. Langage d'assemblage

4. LANGAGES EVOLUES

Un progrès très important dans la communication Homme Machine a été réalisé par la mise en oeuvre des **langages évolués, plus proches du langage humain (Fig. A-4).**
Ce type de langage ne sert plus seulement à exprimer les ordres du programmeur à la machine mais surtout lui permet de **résoudre des problèmes complexes, de manière rigoureuse et contrôlable.**

Les langages de programmation évolués sont très nombreux mais quelques-uns seulement sont d'un emploi courant. On distingue :

- Les langages scientifiques comme **FORTRAN, ALGOL ou BASIC.**

- Les langages orientés gestion comme **COBOL et ses dérivés.**

- Les langages dits universels comme **PL1, APL et PASCAL.** Le langage PASCAL a donné naissance à **MODULA, ADA et C.**

- D'autres langages répondent à des besoins spécifiques : **LISP et PROLOG** manipulent des structures de listes, tandis que **SMALLTALK** a introduit la programmation objet, aujourd'hui relayé par **C++.**

Pour être compris par le processeur, les programmes écrits en langage évolué, nécessitent une traduction en langage binaire.
Il fallait donc concevoir un programme traducteur et l'écrire en langage machine de telle sorte que le travail de traduction soit confié à l'ordinateur lui-même.

Il existe 2 types de programmes traducteurs : le compilateur et l'interpréteur.

Le compilateur considère comme donnée, l'ensemble du programme source et produit en résultat le programme objet en langage machine. La traduction est globale et l'exécution est séparée de la phase de compilation.

L'interpréteur ne traduit le programme source qu'instruction par instruction. Les instructions machines correspondantes sont alors exécutées puis l'interpréteur poursuit par la traduction de l'instruction évoluée suivante. La traduction est donc partielle et l'exécution est effectuée parallèlement à l'interprétation.

Le programme SOURCE écrit en langage évolué, est traduit par le COMPILATEUR ou l'INTERPRETEUR, qui va fournir le code EXECUTABLE écrit en binaire.

L'utilisation d'un langage évolué présente également un autre intérêt. Pour réutiliser, sur un nouvel ordinateur, les programmes déjà écrits dans un langage évolué, il n'est plus nécessaire de réécrire l'ensemble des programmes source, mais seulement d'écrire un programme traducteur correspondant au nouvel ordinateur. **La portabilité des programmes est donc assurée sur des machines différentes.**

Ainsi, le programmeur s'est libéré d'un grand nombre de contraintes liées au dialogue direct avec la machine.

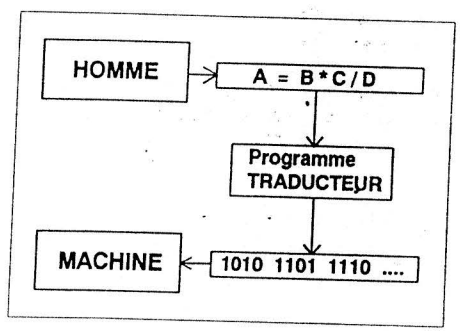


Fig. A-4. Langage évolué

II – ALGORITHMIQUE

1. DEFINITION

A un niveau encore supérieur et plus proche de l'Homme, il apparaît nécessaire, à partir de l'énoncé d'un problème, de réaliser dans un premier temps un "Schéma" rigoureux de résolution. Ce "Schéma", appelé algorithme, est repris dans un second temps (Fig. A–5), pour être programmé dans n'importe quel langage.

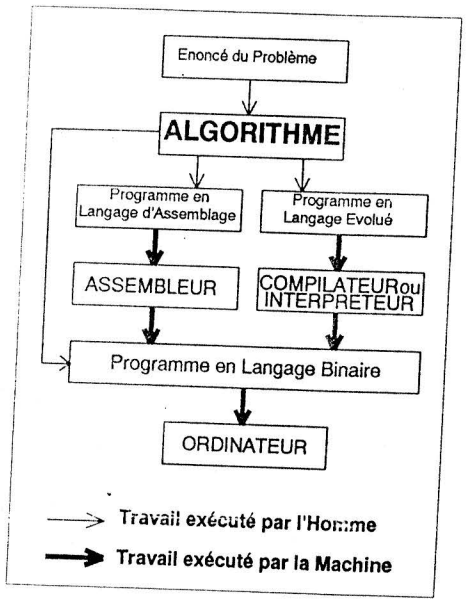


Fig. A–5. Communication Homme Machine

L'algorithmique est l'étude des processus de traitement. Les algorithmes permettent de représenter des schémas de programmation pour résoudre des problèmes complexes dans les domaines administratifs, comptables, scientifiques, industriels.

Pour indiquer à une personne le chemin à suivre pour parvenir à l'endroit demandé, on remarque :
– qu'il est nécessaire de connaître le lieu même où l'on se trouve (on ne peut fournir le renseignement si l'on est soi même perdu). C'est la connaissance de l'***état initial*** ;
– que le demandeur doit être capable de ***réaliser un certain nombre d'actions*** (reconnaître la droite et la gauche, lire le nom des rues ...) ;
– qu'il est nécessaire de connaître l'endroit auquel la personne désire se rendre : C'est la connaissance de l'***état final***.
Le processeur (humain dans le cas présent) doit comprendre et exécuter un certain nombre d'instructions.

Un algorithme est un ensemble de règles opératoires rigoureuses, ordonnant à un processeur particulier d'exécuter dans un ordre déterminé un nombre fini d'opérations élémentaires, pour résoudre tous les problèmes d'un type donné.

Les opérations élémentaires auxquelles se réfère cette définition sont données pour un processeur bien spécifique possédant un jeu d'opérateurs bien défini.

La description des procédés de calculs arithmétiques ont fourni les premiers algorithmes :

Pour un processeur connaissant les opérations d'addition et de soustraction, mais pas l'opération de division entière, cette opération peut être programmée comme une série de soustractions. Cet énoncé doit d'abord être traduit sous une forme littérale indiquant comment résoudre le problème. Il nous servira de base pour illustrer les représentations des algorithmes.

Diviser 14 par 4, revient à :
.Soustraire 4 de 14 *(quotient = 1), (reste = 10)*
.Soustraire 4 du reste 10 *(quotient = 1+1), (reste = 6)*
.Soustraire 4 du reste 6 *(quotient = 1+1+1), (reste = 2)*
.Constater que l'on ne peut plus soustraire 4 du reste 2.
.Ecrire : Le quotient est 3 et le reste 2.

Le problème peut être représenté par les objets **dividende, diviseur, quotient et reste**, que nous repèrons par les **identificateurs de variables *dvdende, dvseur, qtient* et *reste*.**

Etat initial :
dvdende = 14, dvseur = 4, qtient = 0, reste = 14.

Actions à réaliser :
Si *reste > dvseur*
alors
qtient devient *qtient + 1*
reste devient *reste – dvseur.*
Il faudra recommencer cette action dont la condition d'arrêt est *reste < = dvseur.*

Etat final :
dvdende = 14, dvseur = 4, qtient = 3, reste = 2.

La représentation des algorithmes est réalisée soit par un **organigramme**, soit au travers d'un **langage d'expression algorithmique.** Il existe plusieurs méthodes de représentation par organigramme. La méthode suivante est la plus répandue.

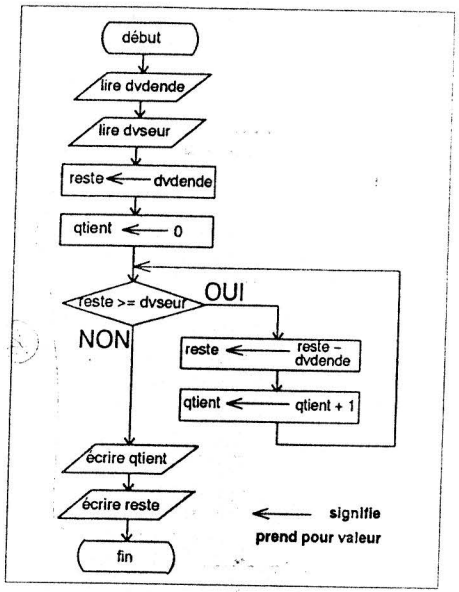


Fig. A–6. Organigramme division entière

2. ORGANIGRAMME

Dans un organigramme (Fig. A–6), les traitements sont représentés par un **graphe fléché et fermé permettant de suivre l'ordre d'exécution.**

L'organigramme utilise des **symboles normalisés** pour illustrer le cheminement du raisonnement logique.

Début ou **fin** de l'organigramme

Entrée Sortie réalisant, soit la lecture d'une donnée, soit son écriture.

Embranchement résultant d'une condition posée sur une donnée.

Traitement composé d'une ou de plusieurs opérations sur des données.

Cette représentation respectant la **norme NF Z 67–010**, est utilisée pour analyser un problème. Elle présente l'**intérêt d'une visualisation globale.** De ce fait, elle ne peut correspondre aux besoins d' études complexes et détaillées, et reste limitée aux algorithmes simples.

3. LANGAGE D'EXPRESSION ALGORITHMIQUE

Le langage d'expression algorithmique fait appel à un **ensemble de mots clés et de structures permettant d'organiser l'exécution des traitements de manière systématique.**

L'algorithme de la division entière est présenté ci-dessous en langage d'expression algorithmique.

```
Début algorithme division entière
  entier : dvdende,dvseur,qtient,reste
  Début action
    écrire 'saisir dividende '
    lire dvdende
    écrire 'saisir diviseur '
    lire dvseur
    reste <--- dvdende
    qtient <--- 0
    Tant que reste >= dvseur faire
      reste <--- reste - dvseur
      qtient <--- qtient + 1
    Fin Tant que
    écrire 'quotient = ', qtient
    écrire 'reste = ',reste
  Fin action
Fin algorithme
```

La syntaxe algorithmique utilisée est stricte. Les mots clés sont soulignés.

L'algorithme ci-dessus donne un aperçu du langage. L'objectif n'est pas d'en détailler la syntaxe, qui sera étudiée dans cet ouvrage, mais d'en comprendre globalement le fonctionnement, exposé à la suite.

■ FONCTIONNEMENT DE L'ALGORITHME

La première ligne de l'algorithme comporte les mots soulignés Début algorithme, suivis du nom *division entière*, donné à l'algorithme par l'analyste du problème.

La deuxième ligne permet de déclarer les variables qui seront utilisées et de définir le domaine dans lequel elles prendront leurs valeurs. Dans l'exemple, les variables *dvdende, dvseur, qtient* et *reste* auront des valeurs de type entier, c'est à dire dans le domaine des entiers.

L'action principale débute par les mots Début action, et comprend plusieurs actions :

écrire est l'action provoquant l'écriture à l'écran de la chaîne de caractères comprise entre les apostrophes.
lire est l'action qui attend de l'utilisateur une saisie au clavier. L'utilisateur donnera ainsi des valeurs aux variables *dvdende* et *dvseur.*

le symbole <--- signifie que la variable située à gauche du symbole prend pour valeur la valeur de l'expression située à droite du symbole. Dans le cas présent , la variable *reste* prend la valeur de la variable *dvdende* et *qtient* prend la valeur 0.

L'action Tant que est une structure répétitive dont le contenu est exécuté si la condition qui suit le Tant que est vraie. Les actions contenues entre Tant que et Fin Tant que modifient les valeurs de *reste* et *qtient* et sont exécutées Tant que *reste* est supérieur ou égal à *dvseur.*

Enfin, les actions écrire affichent les résultats suivants à l'écran :
quotient =, suivi de la valeur de *qtient*
reste =, suivi de la valeur de *reste.*

L'action principale est terminée par les mots soulignés Fin action et l'algorithme par les mots soulignés Fin algorithme.

4. VARIABLES ET TRACE DE L'ALGORITHME

La notion de variable X en Mathématique, écrite par exemple sous la forme $2X^2 - 3X + 1 = 0$, fait référence à une valeur inconnue au moment où le problème est posé, mais existante puisque cette valeur fait partie de la solution du problème. Cette valeur est figée pendant la résolution du problème.

En Informatique, cette notion est différente car **la valeur de la variable peut évoluer pendant la résolution d'un problème.** La nouvelle valeur de la variable venant remplacer ou "écraser" l'ancienne valeur.

La trace d'un algorithme permet de comprendre comment va "tourner" le programme mais aussi de s'assurer qu'il fournit, dans un cas particulier, une solution juste. Cette trace ne donne pas la preuve que l'algorithme est juste, mais indique, dans le cas particulier traité, qu'il aboutit à une solution correcte. Dans ce cas, la trace n'est qu'une présomption de justesse.

■ TRACE DE L'ALGORITHME DIVISION ENTIERE

dvdende	*dvseur*	*reste*	*qtient*
14	4	14	0
14	4	10	1
14	4	6	2
14	4	2	3

Le tableau ci-dessus montre l'évolution des variables mises en oeuvre dans l'algorithme, lorsque l'utilisateur saisit les valeurs 14 pour *dvdende* et 4 pour *dvseur.*

Le lecteur peut ainsi vérifier le déroulement pas à pas de l'algorithme, en le faisant "tourner" à la main.

III – CHOIX DU LANGAGE PASCAL

1. RAISONS DU CHOIX

S'inspirant du langage Algol, **Niklaus Wirth** créa le Pascal et présenta un premier compilateur en 1970.

L'auteur avait clairement énoncé ses objectifs : Le langage devait être **"adapté à l'enseignement de la programmation rigoureuse et structurée"**. Un programmeur Pascal va donc prendre de bonnes habitudes de programmation.

Le Pascal apporte au programmeur la possibilité de définir ses propres **structures de données** à partir des types fondamentaux du langage. Ce langage est donc suffisamment riche pour aborder les domaines scientifique ou industriel, comme celui de la Gestion.

A la fois **langage de programmation et langage d'expression algorithmique**, le Pascal est un langage clair, sans ambiguïté et fournissant des programmes très lisibles. Il est parfaitement adapté à son orientation initiale d'apprentissage de la programmation. Le programme suivant a pour seul but de découvrir un premier programme Pascal pour montrer les similitudes avec l'algorithme établi plus haut.

```
PROGRAM divisentiere (INPUT,OUTPUT);
VAR
  dvdende,dvseur,qtient,reste : INTEGER;
BEGIN
  WRITE('saisir dividende ');
  READLN(dvdende);
  WRITE('saisir diviseur ');
  READLN(dvseur);
  reste := dvdende;
  qtient := 0;
  WHILE reste >= dvseur DO
    BEGIN
      reste := reste - dvseur;
      qtient := qtient + 1
    END;
  WRITELN('quotient = ', qtient);
  WRITELN('reste = ',reste)
END.
```

Langage Pascal et Algorithmique sont intimement liés. Cet ouvrage présente l'algorithmique et le langage Pascal, avec des applications programmées, précédées des algorithmes correspondants.

2. STRUCTURE DU LANGAGE

Tout langage a pour base un alphabet à partir duquel des mots sont formés pour établir des phrases ou instructions.

L'ALPHABET du langage Pascal est composé des éléments suivants :

* Lettres 'a', 'b', ... , 'z', 'A', 'B', ... , 'Z'
* Chiffres 0, 1, 2, ... , 9
* Caractères spéciaux opérateurs + - * / ^ > < = >= <= := séparateurs *espace* ( ) [ ] { } signes de ponctuation . .. , ; : '

D'autres caractères sont utilisés, spécifiques à telle ou telle version du langage.

LES MOTS du langage se répartissent en plusieurs classes :

* **Les mots clés et mots réservés** du langage ont une signification précise pour le compilateur. Pour les distinguer, nous les écrirons en majuscules dans les programmes.
Exemple : READ en Pascal, équivalent de INPUT en Basic, permet de lire une donnée.

* **Les identificateurs** correspondent aux noms donnés par le programmeur aux objets, variables ou constantes, manipulés dans le programme. Ils peuvent être très explicites. Nous les écrirons en minuscules dans les programmes.

Exemples d'identificateurs : *Abcis, nomeleve* en Pascal, équivalent de *X, $NOM* en Basic.

*** Les nombres** permettent de définir des entiers, des réels et peuvent être représentés en notation scientifique.
exemples : 1992 , 3.14 , 0.437E+2.

*** Les chaînes de caractères** sont utilisées pour stocker des noms, des messages ...
exemples : 'MOTEUR' , 'Contact etabli'.

LES PHRASES OU INSTRUCTIONS adoptent une syntaxe pour décrire les données, les traitements et les résultats.

*** Des structures de données permettent de représenter les variables du monde réel :**
- Variables de type numérique, caractère ou logique (Vrai ou Faux),
- Variables de type structuré telles que vecteur, tableau, enregistrement, ensemble, ...

*** Pour traiter les données, le langage Pascal met à la disposition du programmeur des macro-instructions à échelle humaine :**
- Lecture de données saisies au clavier, ou sur un autre organe d'entrée.
- Affectation de valeurs aux variables et Calculs sur des expressions.
- Ecriture des résultats, sur un écran, ou sur un autre organe de sortie.
- Structures de Programmation.

DES STRUCTURES DE PROGRAMMATION de différentes formes représentent la variété des raisonnements humains :

*** Structures linéaires**, pour des instructions exécutées en séquence.

*** Structures conditionnelles**, pour que certaines actions ne soient exécutées que si une condition est satisfaite.

*** Structures répétitives**, pour exécuter des groupes d'instructions POUR un nombre de fois défini, ou TANT QUE une condition est vraie, ou JUSQU'A ce qu'une condition soit satisfaite.

*** Structures puissantes de sous-programmes, procédures et fonctions**, améliorant la lisibilité des programmes, leur maintenance, et la non redondance d'écriture de certaines actions.

3. NORMALISATION DU PASCAL

Devant les succès du Pascal, des extensions ont été rajoutées, introduisant ainsi de nouvelles versions pour en augmenter la puissance et l'efficacité. Les développements les plus connus ont été fournis par :

- l'Université de Californie de San Diégo qui proposa le **Pascal UCSD**, répandu sur micro-ordinateurs,

- Microsoft qui génèra le **MS-Pascal** en 1980, version adoptée par IBM,

- L'éditeur de logiciels Borland qui créa le **Turbo-Pascal**, spécifiquement adapté aux micro-ordinateurs, avec un environnement intégré pour la programmation (éditeur de texte, compilateur, exécution). Il est à noter que Borland poursuit encore l'enrichissement du langage au travers de nouvelles versions,

- On peut également citer le **Pascal 1000** de Hewlett-Packard ou le **Pascal 6000** sur système Control Data.

La **Norme Française Z65-300** sur le langage Pascal a été éditée par l'AFNOR en 1984. Elle correspond à la version française de la **norme Internationale ISO 7185.**

Cet ouvrage présente le Pascal défini par ces normes, équivalent au Pascal Standard défini par l'Auteur.

■ REFERENCES

1. Rapport Pascal (N.WIRTH 1978).
2. Pascal Manuel de l'Utilisateur (K.JENSEN et N.WIRTH 1978).
3. Langage de Programmation Pascal (AFNOR 1984).

B – ALGORITHMIQUE ET PASCAL

I – BASES DES LANGAGES ALGORITHMIQUE ET PASCAL

1. CARACTERISTIQUES

A – STRUCTURE D'UN PROGRAMME – REPRESENTATIONS SYNTAXIQUES

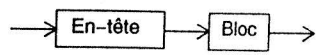
Un programme Pascal se compose d'un en-tête et d'un corps appelé bloc.

<programme> ::= <en-tête> <bloc>

Cette notation, appelée **Notation BNF (Backus-Naur Form)**, permet de décrire totalement la syntaxe du langage.

Un autre mode de représentation du langage est celle des **Diagrammes syntaxiques de Conway.** Ces diagrammes illustrent les possibilités de construction syntaxique des instructions, ainsi que leurs assemblages pour former un programme. Les 2 modes de représentation syntaxique seront utilisés dans cet ouvrage.

Diagramme syntaxique d'un programme



■ REGLES DE LA NOTATION BNF

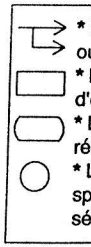
La notation BNF utilise 6 symboles qui lui sont propres. Ils sont à distinguer des autres caractères qui sont ceux du langage Pascal.

Les symboles BNF :

< et >	encadrent une entité,
::=	signifie "s'écrit comme",
|	signifie "ou bien",
{ et }	mentionnent une éventuelle répétition de leur contenu.

■ REGLES DES DIAGRAMMES SYNTAXIQUES

L'écriture des diagrammes syntaxiques de Conway est représentée au travers de 4 formes graphiques : la ligne fléchée, le rectangle, l'ovale et le cercle.



* Les lignes fléchées donnent le sens du ou des chemins de lecture.
* Les rectangles contiennent des noms d'entité définis ultérieurement.
* Les ovales mentionnent les mots réservés du langage Pascal.
* Les cercles mentionnent les caractères spéciaux du langage Pascal, opérateurs, séparateurs et signes de ponctuation.

Dans un bloc, on distingue plusieurs parties dont certaines sont obligatoires alors que d'autres peuvent être vides.

<bloc> ::=
<partie déclaration d'étiquette>
<partie définition de constante>
<partie définition de type>
<partie définition de variable>
<partie procédure et fonction>
<partie instructions exécutables>

Après avoir défini l'en-tête d'un programme, nous étudierons les différentes parties constitutives d'un bloc.

Plutôt que de respecter l'ordre imposé dans la définition du bloc dont certaines parties sont facultatives, il paraît souhaitable de présenter d'abord la partie obligatoire instruction, puis les parties étiquette, constante, variable et enfin la partie type.
Les parties de procédures et de fonctions seront introduites ultérieurement.
Définissons tout d'abord l'en-tête d'un programme.

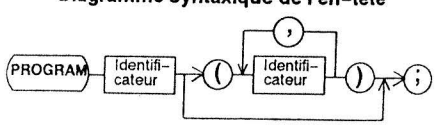
B - EN-TETE DE PROGRAMME

L'En-tête identifie le programme par un nom donné par le programmeur et fournit la liste des fichiers paramètres, extérieurs au programme, qui échangent des informations avec les organes d'entrée et de sortie. L'En-tête ne comprend qu'une seule ligne, commençant par le mot réservé **PROGRAM**. **Un En-tête de programme correspond au Début d'algorithme.**

<en-tête> ::=
PROGRAM *<identificateur>*
(<fichiers paramètres>);

<fichiers paramètres> ::=
<identificateur> {, <identificateur> }

Diagramme syntaxique de l'en-tête



Exemple d'en-tête :

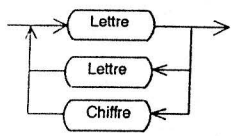
```
Début algorithme essai1
PROGRAM essai1 (INPUT,OUTPUT);
```

Un **identificateur** est une chaîne de caractères que le programmeur va créer pour désigner une constante, une variable, un type de données ou tout autre élément défini par lui : ***essai1*** est l'identificateur du programme.

Certains identificateurs sont prédéfinis dans le langage et ne nécessitent pas de déclaration, ils sont prédéclarés. Ainsi, **INPUT** et **OUTPUT** sont des identificateurs de fichiers standards. INPUT désigne le **clavier** comme fichier d'entrée prédéfini et OUTPUT l'**écran** comme fichier de sortie prédéfini. Ils sont implicites pour l'algorithme (et pour certaines versions de Pascal).

<identificateur> ::=
<lettre> { <lettre> | <chiffre> }

Diagramme syntaxique d'un identificateur



Un identificateur **commence par une lettre, suivi** d'une lettre ou d'un **chiffre. Il ne doit pas contenir** d'espace. Les caractères accentués sont interdits en Pascal standard.

Exemples d'identificateurs :

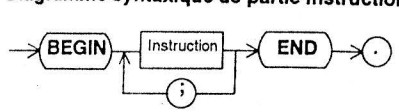
CORRECT	INCORRECT
nompiece	nom piece
prenom1	1prenom
element	élément

C - PARTIE INSTRUCTIONS EXECUTABLES

La partie instruction du programme commence par le mot réservé **BEGIN** et finit par le mot **END .** (suivi d'un point). **En algorithmique, ces mots** sont notés **Début action** et **Fin action**. Les instructions réalisent des actions, qui peuvent être simples ou structurées.

<partie instruction> ::=
BEGIN *<instruction> { ; <instruction> }* END.

Diagramme syntaxique de partie instruction



Exemple de partie instruction :

L'algorithme suivant ne réalise aucune action. Le nom *essai2* est donné au niveau de l'en-tête, suivi de la partie action, qui est vide.

```
Début algorithme essai2
  Début action
  Fin action
Fin algorithme
```

Le programme correspondant ci-dessous montre une partie instruction vide ; elle ne comporte aucune instruction.

```
PROGRAM essai2 (INPUT,OUTPUT);
BEGIN
END.
```

D - PARTIE DECLARATION D'ETIQUETTE

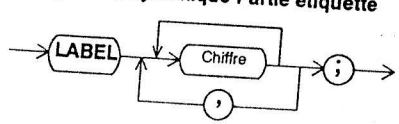
Une étiquette peut être utilisée dans un programme pour faire référence à une instruction. On accède alors à cette instruction par le mot réservé GOTO, suivi du repère d'étiquette.

Une étiquette doit, préalablement à son utilisation, être définie dans la partie déclaration d'étiquette.

Une étiquette est déclarée par le mot réservé **LABEL** suivi d'un entier de 1 à 4 chiffres.

< partie déclaration d'étiquette> ::=
LABEL *<étiquette> { , <étiquette> } ;*

Diagramme syntaxique Partie étiquette



Exemple de déclaration d'étiquette :

```
PROGRAM essai3 (INPUT,OUTPUT);
LABEL 1,2;
BEGIN
END.
```

Le langage Pascal permet de se dispenser des étiquettes et donc de l'instruction **GOTO** qui est indispensable dans d'autres langages. L'utilisation de ces instructions nuit à la lisibilité du programme et conduit à de mauvaises habitudes de programmation. Certains auteurs en parlent comme des "Instructions maudites", à éviter absolument dans la programmation en Pascal.

Les étiquettes et GOTO n'existent pas en algorithmique.

E - PARTIE DECLARATION DES CONSTANTES

■ LES OBJETS CONSTANTES

Les objets mis en oeuvre dans un programme définissent les éléments du problème à traiter. Chaque objet représente un élément du système.

Un objet est défini par son identificateur, son type, et sa valeur.

- **L'identificateur** de l'objet est le nom donné à l'objet par le programmeur dans le programme.

- **Le type** d'un objet est l'ensemble des valeurs possibles que pourra prendre cet objet. Le Langage Pascal définit des types standards sur lesquels nous reviendrons. Ces types sont dits prédéclarés. Ils sont donnés dans l'encadré suivant.

TYPES SIMPLES STANDARDS

- **INTEGER** pour un nombre **entier**,
- **REAL** pour un nombre **réel**,
- **CHAR** pour un **caractère** (lettre ou un chiffre ou caractère spécial),
- **BOOLEAN** pour une **valeur logique** (elle ne peut prendre que les valeurs **TRUE** pour VRAI et **FALSE** pour FAUX).

- **La valeur** des objets peut évoluer ou non au cours de la résolution du problème. Les objets dont la valeur ne varie pas sont des constantes. Les autres objets sont des variables. Elles seront étudiées dans le point F.

Exemple de définition de CONSTANTE :

L'objet π est une constante à laquelle on associe l'identificateur *pi* ; le type est réel ; la valeur est figée à 3.14 (Un point remplace la virgule décimale).

Autres exemples :

Dans l'encadré suivant, les premières lignes correspondent au libellé complet des objets dans le langage naturel, et les secondes lignes définissent l'identificateur, le type et un exemple de la **valeur figée** de chaque objet.

Charge maximale de l'ascenseur		
chargemax	INTEGER	**800**
Accélération de la pesanteur		
g	REAL	**9.81**
Masse atomique de l'oxygène		
MO2	INTEGER	**16**
Capacité de la cuve numéro 1		
CAPACUVE1	REAL	**35000**
Réponse négative		
negatif	CHAR	**'N'**

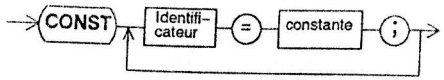
Remarque 1 : La chaîne de caractères correspondant à l'identificateur ne doit pas contenir d'espace : *charge max* n'est pas un identificateur correct. Rappelons également qu'un identificateur ne contient pas de caractère accentué et qu'il doit commencer par une lettre.

Remarque 2 : Sur micro-ordinateur, une variable de type INTEGER prend ses valeurs dans l'ensemble [–32768,+32767]. Pour cette raison, la constante *CAPACUVE1* a été déclarée de type réel. La constante **MAXINT** est une constante prédéclarée entière qui vaut 32767.

■ SYNTAXE PARTIE DES CONSTANTES

<partie définition de constante> ::= *<vide>* | **CONST** *<définition de constante>* *{ ; <définition de constante> } ;*
<définition de constante> ::= *<identificateur> = <constante>*
<constante> ::= <nombre> | <caractères>

Diagramme de la partie des constantes



Exemple de déclaration de constantes :

L'algorithme ci-dessous utilise les types entier, réel et caractère. Il définit une constante, sur la ligne du type correspondant, en donnant par le signe d'égalité, une valeur à un identificateur. Les constantes sont séparées par des virgules.

```
Début algorithme essai4
   entier : chargemax = 800, MO2 = 16
   réel : g = 9.81, capacuve1 = 35000
   caractère : negatif = 'N'
   Début action
   Fin action
Fin algorithme
```

Le programme correspondant à l'algorithme essai4 s'écrit sous la forme suivante :

```
PROGRAM essai4 (INPUT,OUTPUT);
CONST
   chargemax =800; g =9.81;
   MO2 = 16; capacuve1 = 35000;
   negatif = 'N';      (* voir remarque *)
BEGIN
END.
```

La partie instruction de ce programme est vide. Les constantes sont définies en début de programme dans la partie de déclaration des constantes, située en début de bloc, après une éventuelle partie étiquette.

Remarque : Certaines versions de Pascal n'admettent pas la notation *capacuve1 = 35000*, parce que cette valeur est supérieure à MAXINT (32767). Ces versions nécessitent l'utilisation de la notation scientifique des réels (35000 s'écrit alors 0.35E+05, soit 0.35 multiplié par 10 puissance 5), qui est développée PAGES 21 et 22.

EXERCICE I–1 : Définir un autre objet CONSTANTE de chaque type, avec identificateur et exemple de valeur.

EXERCICE I–2 : Ecrire un algorithme et un programme déclarant ces CONSTANTES. La partie instruction sera vide.

F – PARTIE DECLARATION DES VARIABLES

■ LES OBJETS VARIABLES

Nous avons vu dans la définition des objets CONSTANTES qu'il existe aussi des objets VARIABLES.

Un objet VARIABLE est également défini par son identificateur, son type, et sa valeur.

Nous ne revenons pas sur les définitions de l'identificateur, du type et de la valeur d'un objet, qui ont été étudiées dans la partie Déclaration des Constantes. Rappelons simplement qu'un objet VARIABLE possède une valeur qui peut évoluer, dans un certain domaine appelé type de l'objet, au cours de la résolution du problème.

Exemple de définition de VARIABLE :

L'objet nombre de tours par minute est une variable pouvant avoir comme identificateur *tourmin* ; le type est entier ; et nous donnons un exemple de valeur : 2500.

Autres exemples :

Dans l'encadré suivant, les premières lignes correspondent au libellé complet des objets dans le langage naturel, et les secondes lignes définissent l'identificateur, le type et un exemple de **valeur non figée** de chaque objet.

Lettre de l'alphabet		
caract	CHAR	**'b'**
Niveau bas atteint		
niveaubas	BOOLEAN	**FALSE**
A est plus grand que B (A > B)		
asupb	BOOLEAN	**TRUE**
Hauteur de l'ascenseur en mètres		
hteur	REAL	**5.72**
Vitesse de l'ascenseur en m/s		
vitesse	REAL	**1.2**
Temps écoulé		
temps	REAL	**6.35**
Numéro de l'étage		
etage	INTEGER	**2**

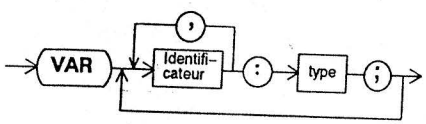
Remarque 1 : Une variable de type réel peut également s'écrire avec la notation scientifique 123.456E2 qui se lit 123.456 multiplié par 10 puissance 2. Nous reviendrons sur ces notions dans le paragraphe traitant du formatage des écritures.

Remarque 2 : Une variable de type BOOLEAN (logique) ne peut prendre que les valeurs logiques TRUE pour VRAI ou FALSE pour FAUX.

■ SYNTAXE PARTIE DES VARIABLES

<partie définition de variable> ::= *<vide>* | **VAR** *<définition de variable>* *{, <définition de variable> } ;*
<définition de variable> :: = *<identificateur> { ,<identificateur> } : <type>.*

Diagramme de la partie des variables



Exemple de déclaration de variables :

L'algorithme *ascenseur* peut s'écrire sous la forme suivante :

```
Début algorithme ascenseur
   entier : chargemax = 800 , etage
   réel : temps, hteur, vitesse, g = 9.81
   caractère : caract, negatif = 'N'
   logique : niveaubas
Début action
Fin action
Fin algorithme
```

Cet algorithme utilise les types entier, réel caractère et logique (ou booléen). Il déclare une variable par son identificateur, sur la ligne du type correspondant. Constantes et variables peuvent coexister sur une même ligne.

Le programme correspondant à l'algorithme *ascenseur* est présenté à la page suivante.

```
PROGRAM ascenseur (INPUT, OUTPUT);
CONST
   chargemax = 800; g = 9.81;
   negatif = 'N';
VAR
   temps,hteur,vitesse : REAL;
   etage : INTEGER;
   niveaubas : BOOLEAN;
   caract : CHAR;
BEGIN
END.
```

La partie instruction de ce programme est vide. Les variables sont définies en début de programme dans la partie de déclaration des variables, située en début de bloc, après la partie déclaration de Constantes. La partie déclaration des Types, traitée au point G, n'apparaît pas.

EXERCICE I-3 : Définir un autre objet VARIABLE de chaque type, avec identificateur et exemple de valeur.

EXERCICE I-4 : Ecrire un algorithme et un programme déclarant ces VARIABLES. La partie instruction sera vide.

G – PARTIE DECLARATION DE TYPE.

On distingue les types simples avec lesquels une variable ne possède qu'une seule valeur, et les types structurés (définis et utilisés plus tard) avec lesquels une variable dite structurée possède une collection de valeurs.

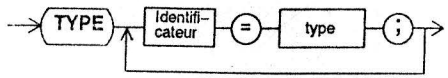
Nous avons déjà utilisé les types standards **INTEGER, REAL, CHAR et BOOLEAN** qui sont des types simples prédéclarés.

TYPES SIMPLES STANDARDS	
INTEGER	pour un nombre **entier**,
REAL	pour un nombre **réel**,
CHAR	pour un **caractère**
BOOLEAN	pour une **valeur logique** (valeurs **TRUE** pour VRAI et **FALSE** pour FAUX).

Le programmeur peut définir et déclarer de nouveaux types tels que le **type scalaire** ou le **type intervalle**.

< partie déclaration de type> ::=
<vide> | **TYPE** *<identificateur> = <type>;*

Diagramme syntaxique partie des types

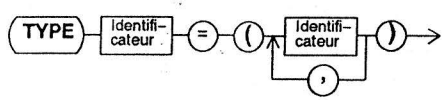


■ LE TYPE SCALAIRE

Le type scalaire est un autre type simple que le programmeur peut lui-même définir, comme un **ensemble de valeurs énumérées,** sur lequel s'applique une relation d'ordre.

<type scalaire> ::=
(<identificateur> { , <identificateur> })

Diagramme syntaxique du type scalaire



Une variable ayant le type *metal* défini ci-dessous ne pourra prendre ses valeurs que dans l'ensemble des valeurs énumérées.

```
TYPE metal = (plomb, etain, zinc, aluminium,
              cuivre, fer);
```

Exemple de déclaration de type scalaire :

```
PROGRAM essai5 (INPUT,OUTPUT);
TYPE metal = (plomb, etain, zinc,
              aluminium, cuivre, fer);
VAR matiere : metal;
BEGIN
END.
```

Remarque : Contrairement au type REAL, les types INTEGER, CHAR et BOOLEAN sont des types scalaires.

Notation algorithmique de la variable *matiere*

énumération (plomb,etain,zinc, aluminium,cuivre,fer) : matiere

Les opérateurs relationnels = <> < > <= >= ainsi que les trois fonctions spécifiques suivantes s'appliquent à toutes les valeurs d'un type scalaire standard (CHAR, INTEGER, BOOLEAN) ou d'un type scalaire ou intervalle défini par le programmeur.

SUCC ( )	pour successeur,
PRED ( )	pour prédécesseur,
ORD ( )	pour rang.

Exemples : Fonction	Valeur
SUCC(etain)	zinc
PRED(cuivre)	aluminium
ORD(zinc)	2 (plomb a le rang 0)
SUCC('C')	'D'
PRED('B')	'A'
SUCC(15)	16

La relation d'ordre s'appliquant sur les caractères permet d'écrire **'D' < 'E'**. Cet ordre s'applique également sur les constantes booléennes avec **FALSE < TRUE** (FAUX < VRAI).

La fonction CHR( ) n'admet comme paramètre qu'un nombre entier. Cette fonction fournit comme résultat le caractère dont le numéro d'ordre est spécifié comme paramètre.
Les fonctions **CHR( ) et ORD( )** sont des fonctions inverses telles que :

CHR(ORD('A')) = 'A'

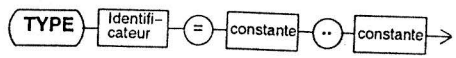
EXERCICE I-5 : Définir un autre type scalaire et l'intégrer dans la partie déclaration de type d'un programme.

■ LE TYPE INTERVALLE

Le type intervalle est un **sous ensemble de type scalaire** déjà défini incluant tous les éléments compris entre les bornes mentionnées.

<type intervalle> ::=
<constante>..<constante>

Diagramme syntaxique type intervalle



Une variable du type *heure* est bornée par les valeurs 0 et 24. Le type *heure* est un sous-ensemble du type INTEGER.

```
TYPE heure = 0 .. 24;
```

Une variable du type *bassefusion* (défini ci-dessous comme sous-ensemble du type *metal* défini précédemment) pourra prendre les valeurs plomb, etain ou zinc.

```
TYPE bassefusion = plomb .. zinc;
```

Exemple de déclaration de type intervalle :

```
PROGRAM essai6 (INPUT,OUTPUT);
TYPE metal = (plomb, etain, zinc,
              aluminium, cuivre, fer);
     bassefusion = plomb .. zinc;
VAR matiere : metal;
    fondu : bassefusion;
BEGIN
END.
```

Notation algorithmique de la variable *fondu* :

énumération (plomb,etain,zinc, aluminium,cuivre,fer) : matiere
Intervalle plomb .. zinc : fondu

EXERCICE I-6 : Définir un autre type intervalle et l'intégrer dans la partie déclaration de type d'un programme.

2. INSTRUCTIONS SIMPLES

Une instruction réalise une action plus ou moins complexe. On distingue les instructions simples et les instructions structurées.

```
<instruction> ::=
  <instruct° simple> | <instruct° structurée>.
```

L'instruction simple de base est l'affectation. On peut également considérer comme instructions simples, les opérations élémentaires d'entrée sortie correspondant à la lecture de données sur le clavier et à l'écriture de données sur l'écran.

1°– AFFECTATION

L'affectation, notée en algorithmique par le symbole <--- ou **ppv** (prend pour valeur), se traduit en Pascal par := (2 points, immédiatement suivi du signe d'égalité, sans espace intermédiaire).
Cette instruction remplace la valeur courante d'une variable par une nouvelle valeur. La nouvelle valeur peut être donnée directement, ou être une expression fournissant une valeur de même type que la variable.

Les exemples ci-dessous sont des affectations directes de valeur à une variable. D'autres exemples d'affectation par des expressions sont donnés dans le chapitre **3.** , traitant des expressions.

Exemples d'affectation dans l'algorithme

```
tourmin <--- 2500
caract <--- 'b'
hteur <--- 3.1
etage <--- 1
niveaubas <--- VRAI
```

Exemples d'affectation dans le programme

```
tourmin := 2500;
caract := 'b';
hteur := 3.1;
etage := 1;
niveaubas := TRUE;
```

On affecte ou on assigne une valeur à une variable, affectation et assignation sont des termes équivalents. On dira *tourmin* prend pour valeur 2500, ce qui explique la notation abrégée **ppv** en algorithmique.

■ INCREMENTATION

L'incrémentation est une affectation particulière mettant en jeu une seule variable apparaissant à la fois à gauche et à droite du symbole d'affectation. Cette forme d'instruction est largement utilisée en programmation.

```
A <--- A + 1
A := A + 1;
```

Pour bien comprendre comment fonctionne cette instruction, il faut considérer que la machine travaille en 2 temps:

- **Premier temps** : Ignorant totalement la partie à gauche du signe :=, le travail ne porte que sur la partie droite pour calculer la valeur de A + 1,

- **Second temps** : Considérant la variable à gauche du signe :=, la valeur calculée dans le premier temps lui est affectée.

***A* prend donc pour nouvelle valeur**, l'ancienne valeur de *A*, augmentée de une unité, qui est ici la valeur du **pas d'incrémentation.**

Exemple : Si *vitesse* possède la valeur 0.5,
L'incrémentation suivante,
vitesse := vitesse + 0.3
donne à la variable *vitesse* la valeur 0.8

L'incrémentation est plus difficile à appréhender dans les langages utilisant le signe égal à la fois comme opérateur relationnel et comme symbole d'affectation.

2°– LECTURE

La lecture est représentée dans l'algorithme par le mot clé lire et dans le langage Pascal par la procédure prédéfinie **READ** ou **READLN** . La lecture d'une variable communique à cette variable une valeur issue d'un organe d'entrée (clavier ou autre fichier d'entrée).

La syntaxe complète de l'instruction de lecture sera détaillée plus tard, à propos des fichiers.

Dans le cas simple ou la lecture est réalisée au clavier, cette instruction provoque l'attente de la saisie d'une valeur par l'utilisateur.

Exemples de lecture dans l'algorithme

```
lire hteur
lire temps
lire etage, caract
```

Exemples de lecture dans le programme

```
READ(hteur);
READLN(temps);
READ(etage,caract);
```

Une instruction de lecture autorise la lecture d'une ou de plusieurs variables ; les variables sont alors séparées par des virgules. Ainsi, pour la 3ème lecture, l'utilisateur devra saisir 2 valeurs, la première sera donnée à la variable *etage*, et la seconde à la variable *caract*.

La différence entre READ et READLN, correspond au fait que le **LN provoque un saut de ligne après avoir exécuté l'instruction.** (LN signifie Line New c'est à dire Nouvelle Ligne).

3°– ECRITURE

L'écriture est représentée dans l'algorithme par le mot clé écrire et dans le langage Pascal par la procédure prédéfinie **WRITE** ou **WRITELN**. L'écriture communique la valeur d'une variable ou d'une constante à un organe de sortie (écran ou imprimante ou fichier de sortie).

La syntaxe complète de l'instruction d'écriture sera détaillée plus tard, à propos des fichiers.

Dans le cas simple ou l'écriture est réalisée sur l'écran, nous parlerons d'**affichage écran.**

Une instruction d'écriture autorise l'écriture d'une ou de plusieurs variables, constantes ou chaînes de caractères ; ces éléments sont alors séparées par des virgules.

Les exemples suivants montrent comment afficher sur l'écran,
- Une chaîne de caractères,
- La valeur d'une variable,
- Une chaîne de caractère suivie de la valeur d'une variable.

Exemples d'écriture dans l'algorithme

```
écrire 'Mesure de la Vitesse : '
écrire vitesse
écrire 'Numéro de l''étage = ',etage
```

Exemples d'écriture dans le programme

```
WRITE('Mesure de la Vitesse : ');
WRITELN(vitesse);
WRITELN('Numero de l''etage =',etage);
```

Remarque 1 : Les caractères accentués sont seulement utilisés dans les chaînes de caractères des algorithmes ; jamais dans les identificateurs. Pour écrire une **apostrophe** dans une chaîne de caractères, il est nécessaire de la doubler.

Remarque 2 : Les instructions WRITE et WRITELN réalisent toutes deux une écriture à l'écran ; mais **WRITELN provoque un saut de ligne après avoir exécuté l'écriture.**

La première instruction affiche à l'écran la chaîne de caractères 'Mesure de la vitesse', puis la deuxième instruction affiche, à la suite, sur la même ligne, la valeur de la variable *vitesse* . La troisième instruction affiche sur la ligne suivante la chaîne de caractères *'Numero de l''etage'* suivie de la valeur de la variable *etage*.

Affichage écran si vitesse vaut 1.2 et etage 3

```
Mesure de la vitesse 0.12E+1
Numero de l'etage =       3
```

La virgule décimale des nombres réels est remplacée par un point décimal.

La valeur 1,2 devrait donc s'afficher 1.2 mais le Pascal utilise par défaut la notation scientifique pour l'écriture des nombres réels. Les différents formats d'écriture sont étudiés ci-après.

■ FORMATAGES D'ECRITURE

Si A est une variable entière ayant pour **valeur** 123, l'instruction **WRITE(A)**, affichera à l'écran la valeur de A, calée sur la droite d'une zone de 12 à 14 caractères. **La valeur 123 sera donc précédée de 9 à 11 caractères blancs.**

Si B est une **variable réelle** ayant pour **valeur 45.6**, l'instruction **WRITE(B)**, affichera à l'écran la valeur de B en notation scientifique (virgule flottante), sous la forme **0.456E+02** . La mantisse 0.456, est suivie de E, et de l'exposant. Le nombre se lit 0.456 multiplié par 10 puissance 2.
La mantisse est toujours comprise entre 0.1 et 1, dans l'intervalle [0.1 ; 1[.

Les instructions WRITE et WRITELN possèdent **deux paramètres, utilisés pour formater** différemment les écritures des nombres.

WRITE(identificateur : n : m)

```
WRITE( N(A:3);
WRITE(B:5:1);
```

n indique le nombre de caractères réservés pour l'écriture du nombre (y compris le point décimal s'il y a lieu) et **m** définit le nombre de décimales après le point décimal des nombres réels.

Un nombre affichera toujours la totalité de sa partie entière même si **n** est inférieur au nombre de chiffres composant la partie entière du nombre.

Dans l'encadré ci-dessus,
- **l'écriture de A** fournit la valeur 123 non précédée de caractère blanc, soit un total effectif de 3 positions,
- **l'écriture de B** s'affiche sur 5 positions en affichant la valeur 45.6 précédée d'un caractère blanc, soit
  3 positions +
  1 position pour le point décimal +
  1 position après le point.

EXERCICE I-7 : Soit A une variable ayant la valeur 24.531. Ecrire les instructions d'écriture formatée produisant les résultats suivants :
24.5
espace espace 24.5
24.53
espace 24.53
24.531
0.24531E+02

APPLICATION I-1

Algorithme et programme **de saisie et d'écriture formatée** de la variable *référence article*, entière sur 3 positions et de la variable *prix article*, réelle sur 6 positions dont 2 après le point décimal.

a) Algorithme

```
Début Algorithme litecrit
   entier : reference
   réel : prix
Début action
   écrire 'Saisir la référence de l''article'
   lire reference
   écrire 'Saisir le prix de l''article'
   lire prix
   écrire 'Pour l''article ', reference : 3
   écrire 'Le prix est : ',prix :6 :2
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM litecrit (INPUT,OUTPUT);
VAR
   reference: INTEGER;
   prix : REAL;
BEGIN
   WRITE ('Saisir la reference de l''article ');
   READLN (reference);
   WRITE ('Saisir le prix de l''article ');
   READLN (prix);
   WRITELN ('Pour l''article ', reference : 3);
   WRITELN ('Le prix est : ', prix :6 :2)
END.
```

IMPORTANT

– Dans le langage Pascal, deux instructions sont séparées par un point virgule.
– Le mot END n'est jamais précédé par un point virgule.

Remarque : Bien que le langage d'expression algorithmique se situe à un niveau supérieur par rapport à tous les langages de programmation, nous adopterons, dans les algorithmes, les formatages d'écriture des nombres du langage Pascal.

■ COMMENTAIRE

Un commentaire est une explication, donnée par le programmeur, sur une instruction ou un groupe d'instructions.

Dans l'algorithme, on le note entre /* et */.
En Pascal, le commentaire est inscrit entre (* et *) ou entre { et }.

Le commentaire n'est pas pris en compte lors de l'exécution du programme. Le programmeur devra l'utiliser judicieusement pour documenter certaines parties de programme.

Exemple de Commentaire dans l'algorithme

```
Début algorithme commentaire
/* Un 1er commentaire peut indiquer la */
/* nature du problème traité */
   Début action
   /* Un 2ème commentaire peut indiquer la */
   /* méthode utilisée pour le résoudre */
   Fin action
Fin algorithme
```

Exemple de Commentaire dans le programme

```
PROGRAM commente (INPUT,OUTPUT);
(* Un 1er commentaire peut indiquer la *)
(* nature du probleme traite *)
  BEGIN
  (* Un 2eme commentaire peut indiquer la *)
  (* methode utilisee pour le resoudre *)
  END.
```

EXERCICE I-8 : Reprendre l'application I-1 et ajouter un commentaire illustrant la déclaration des variables et un commentaire précédant l'écriture du résultat.

3. EXPRESSIONS

Une expression est une formule qui calcule une valeur. Elle se compose d'**opérateurs** indiquant les opérations à réaliser et d'**opérandes** qui sont les valeurs ou les variables sur lesquelles portent les opérateurs. On distingue les expressions numériques des expressions logiques.

1°- EXPRESSIONS NUMERIQUES

Nous avons rencontré une expression numérique lors de l'étude de **l'incrémentation A := A + 1.**
Cette instruction utilise l'opérateur d'addition pour évaluer l'expression A+1 et affecte la valeur de l'expression A+1 à la variable A.
De la même façon on peut écrire **A := B + C.**
Cette instruction affecte à A la valeur résultat du calcul de l'expression B + C.
Les opérateurs ainsi que les fonctions permettant de construire d'autres expressions sont donnés ci-dessous.

■ LES OPERATEURS

Les opérateurs disponibles pour les expressions numériques sont donnés ci-dessous.

+	addition
–	soustraction
*	multiplication
/	division réelle
DIV	division entière
MOD	modulo

Exemples :

14 DIV 4 donne pour résultat **3.**
14 MOD 4 donne pour résultat **2.**
A est multiple de B si **A MOD B = 0.**

MOD est l'opérateur MODULO permettant d'obtenir le reste de la division entière.
DIV et **MOD n'opèrent que sur des entiers.**

EXERCICE I-9 : Sachant que A = 17 et B = 3, calculer la valeur des expressions suivantes :

(A+B) MOD B	(2)
(A*B) DIV (A–B)	(3)
(A DIV B) + (A MOD B)	(7)
(B MOD A) * B	(9)

■ LES FONCTIONS

Les fonctions suivantes, appartenant au Pascal Standard, admettent des paramètres (ou arguments) numériques.

ABS( )	Valeur absolue
SIN( )	Sinus
COS( )	Cosinus
ARCTAN( )	Arc tangente
SQR( )	Carré (Square)
SQRT( )	Racine carrée (Square Root)
ROUND( )	Arrondi
TRUNC( )	Partie entière
LN( )	Logarithme Népérien
EXP( )	Exponentielle.

Exemples :

Valeurs de A	Fonction	Résultat
A = 7	SQR(A)	49
A = 1.44	SQRT(A)	1.2
A = 6.78	ROUND(A)	7
	TRUNC(A)	6
A = −6.78	ROUND(A)	−7
	TRUNC(A)	−6

EXERCICE I–10 : Sachant que A = 5.6, B = −9, calculer la valeur des expressions suivantes :

SQR(B)	(81)
SQRT(ABS(B))	(3)
ROUND(A+B)	(−3)
TRUNC(A−B)	(14)

■ PRIORITE DES OPERATEURS

Comment sera évaluée l'expression A − B * C ,
comme (A−B)* C ou
comme A − (B*C) ?
La seconde proposition est la bonne.
L'ordre hiérarchique est donné ci-dessous.

1. **Contenu des parenthèses**
2. **Arguments des fonctions**
3. **Résultats des fonctions**
4. **Opérateurs * / DIV MOD**
5. **Opérateurs + −**

Pour 2 opérateurs de même niveau de priorité, les opérations se font de la gauche vers la droite. La modification de cet ordre est réalisable par l'utilisation de parenthèses dont le contenu est évalué en premier.

EXERCICE I–11 : Sachant que A=4, B=−2 et C=5.8, calculer la valeur des expressions suivantes:

(A*B)/(A+B)	(−4)
ABS(B−A)+ROUND(B+C)	(10)
(SQR(B)+C)/ABS(B)	(4.9)
(TRUNC(C−A))*6/(A−B)	(1)

APPLICATION I–2

Algorithme, et programme de **transformation en radians d'une valeur d'angle lue en degrés.** Afficher le résultat avec 4 décimales.

a) Algorithme

```
Début Algorithme degré vers radian
  /* L'utilisateur saisit une valeur en degré */
  /* Le calcul fournit la valeur calculée en radian*/
  réel : d, r,
  /* d pour degré , r pour radian */
  réel : pi = 3.141592
Début action
  écrire 'Saisir la valeur de l''angle en degré '
  lire d
  r <--- d * pi / 180
  écrire 'valeur en radian = ', r :6:4
Fin action
Fin Algorithme
```

Remarque : Les commentaires indiquent la nature du problème, la méthode de résolution et expliquent que la variable ***d*** est utilisée pour les degrés, tandis que la variable ***r*** est utilisée pour les radians.

b) Programme Pascal

```
PROGRAM degreversradian (INPUT,OUTPUT);
(* L'utilisateur saisit une valeur en degre *)
(* Le calcul fournit la valeur calculee en radian *)
CONST
  pi = 3.141592;
VAR
  d, r : REAL;
  (* d pour degre , r pour radian *)
BEGIN
  WRITE ('saisir la valeur de l''angle en degre ');
  READLN (d);
  r := d * pi / 180;
  WRITELN ('valeur en radian = ', r :6:4)
END.
```


■ INITIALISATION

Toute variable doit obtenir une valeur lors de sa première apparition dans le corps du programme. L'initialisation réalise cette action qui se fait soit par une lecture, soit par une affectation simple, soit par une affectation faisant appel à une expression. (Voir Application I–3).

APPLICATION I–3

Déterminer l'intensité d'un courant alternatif donné par la formule i = Imax sin(ωt + ϕ), en fonction du temps (de 0 à 0.02 s) et de l'angle ϕ lu en degré. Afficher le résultat avec 2 décimales.
On donne ω = 314 rad/s et la valeur de la variable imax qui prendra pour valeur 10 A.
a) Construire et commenter le tableau des objets représentant les variables et constantes du problème.
b) Ecrire l'algorithme
c) Ecrire le programme
d) Exemples de calcul.

L'alphabet du langage Pascal ne dispose pas des symboles ω et ϕ. Nous les traduirons par w et phi.

a) Tableau des objets

Objet	Identificateur	Type
Décalage de phase	phi	réel
Facteur de conversion π	pi	réel
Temps en s	t	réel
Pulsation en rad/s	w	réel
Intensité maximale en A	imax	réel
Intensité instantanée en A	i	réel

Les variables saisies par l'utilisateur sont :
– l'angle *phi* lu en degré et qu'il faudra convertir en radian, et
– le temps *t*.
Les autres variables sont :
– l'intensité maximale *imax*, prenant la valeur 10,
– l'intensité calculée *i*.

Les constantes du problème sont :
– Le facteur de conversion π , noté *pi* = 3.14
– la pulsation ω , notée *w* = 314

b) Algorithme

```
Début Algorithme intensité
  réel : t , w = 314, pi = 3.14, imax , phi , i
Début action
  /* Initialisation par affectation */
  Imax <--- 10
  écrire 'Saisir l''angle phi en degré'
  /* Initialisation de phi par lecture */
  lire phi
  écrire 'Saisir le temps en s'
  /* Initialisation de temps par lecture */
  lire t
  phi <--- phi * pi / 180
  i <--- imax * sin(w * t + phi)
  écrire 'intensité instantanée = ', i :4:2
Fin action
Fin Algorithme
```

c) Programme Pascal

```
PROGRAM intensite (INPUT,OUTPUT);
CONST   w = 314; pi = 3.14;          (*constantes*)
VAR     t, i, imax, phi : REAL;      (* variables *)
BEGIN
  Imax := 10;  (* Initialisation par affectation *)
  WRITE('Saisir l''angle phi en degre ');
  READLN(phi);  (* Initialisation par lecture *)
  WRITE('Saisir le temps en s ');
  READLN(t);                         (* lecture *)
  phi := phi * pi / 180;
  i := imax * sin(w * t + phi);      (* calcul *)
  WRITELN('intensite instantanee = ', i :4:2)
END.
```

d) Exemples de calcul

imax	w	phi	t	i
10	314	30	0.000	5.00
10	314	30	0.004	9.78
10	314	30	0.008	1.06
10	314	30	0.012	−9.13
10	314	30	0.016	−6.71
10	314	30	0.020	4.97
10	314	60	0.000	8.66
10	314	60	0.004	7.44
10	314	60	0.008	−4.05
10	314	60	0.012	−9.95
10	314	60	0.016	−2.11
10	314	60	0.020	8.64

...ON I-4

...3 décimales, la ...lues au clavier.

a) Algorithme

```
Début algorithme température moyenne
  réel : tmp1, tmp2, tmp3, moy
  /* températures 1, 2 et 3, et moyenne */
  entier : nbtmp = 3
  /* nombre de températures */
Début action
  écrire 'saisir température 1'
  lire tmp1
  écrire 'saisir température 2'
  lire tmp2
  écrire 'saisir température 3'
  lire tmp3
  moy <--- (tmp1 + tmp2 + tmp3) / nbtmp
  écrire 'la moyenne est ',moy :5:3
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM moy3temp (INPUT,OUTPUT);
CONST
  nbtmp = 3;
VAR
  tmp1, tmp2, tmp3, moy : REAL;
BEGIN
  WRITE ('saisir temperature 1 ');
  READLN (tmp1);
  WRITE ('saisir temperature 2 ');
  READLN (tmp2);
  WRITE ('saisir temperature 3 ');
  READLN (tmp3);
  moy := (tmp1 + tmp2 + tmp3) / nbtmp;
  WRITE ('la moyenne est ',moy :5:3)
END.
```

Remarque : Les variables *tmp1*, *tmp2* et *tmp3* sont initialisées par lecture. La variable *moy* prend pour valeur le résultat de l'expression. L'écriture de la variable *moy* est formatée sur 5 positions, dont 3 après le point décimal.

EXERCICE I-12 : Algorithme et programme calculant la surface S d'un triangle, après saisie de la base B et de la hauteur H (S = 1/2 * B * H).

APPLICATION I-5

Algorithme et programme de conversion d'un temps saisi en secondes par l'utilisateur, en heures, minutes et secondes.

a) Algorithme

```
Début algorithme secondes
  entier : temps, memorise, heur, min, sec
Début action
  écrire 'saisir un temps en secondes'
  lire temps
  heur <--- temps DIV 3600
  memorise <--- temps MOD 3600
  min <--- memorise DIV 60
  sec <--- memorise MOD 60
  écrire temps :5,' en secondes = '
  écrire heur :2,' heures '
  écrire min :2,' minutes ',sec :2,' secondes'
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM secondes(INPUT,OUTPUT);
VAR
  temps, memorise, heur, min, sec : INTEGER;
BEGIN
  WRITELN('saisir un temps en secondes');
  READLN(temps);
  heur := temps DIV 3600;
  memorise := temps MOD 3600;
  min := memorise DIV 60;
  sec := memorise MOD 60;
  WRITE(temps :5,' en secondes = ');
  WRITE(heur :2,' heures ');
  WRITELN(min :2,' minutes ',sec :2,' secondes')
END.
```

Remarque : Cette application met en oeuvre la variable *memorise* qui sert d'intermédiaire dans les calculs. Les écritures des variables *temps*, *heur*, *min* et *sec* sont formatées sur 2 positions.

EXERCICE I-13 : Algorithme et programme calculant le volume d'un cylindre, après saisie de son rayon R et de la hauteur H.
On rappelle V = π * R * R * H.

APPLICATION I-6

Algorithme et Programme calculant la **résistance équivalente de 3 résistances montées en parallèle**, selon le schéma de la figure B1.
On donne 1/R = 1/R1 + 1/R2 + 1/R3.

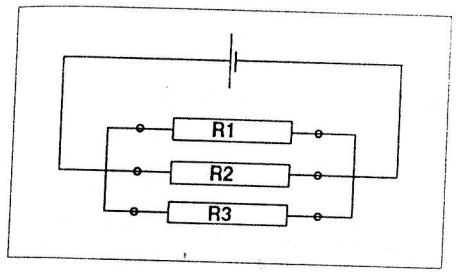


Fig. B-1. Montage en parallèle.

a) Algorithme

```
Début Algorithme résistance
  réel : R, R1, R2, R3, X    /* X pour 1/R */
Début action
  écrire 'Saisir valeurs des 3 résistances'
  lire R1, R2, R3
  /* On calcule X égal à 1/R */
  X <--- 1/R1 + 1/R2 + 1/R3
  /* On calcule R égal à 1/X */
  R <--- 1/X
  écrire 'résistance équivalente = ', R
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM resistance (INPUT,OUTPUT);
VAR
  R, R1, R2, R3 , X : REAL;      (* X pour 1/R *)
BEGIN
  WRITELN('Saisir valeurs des 3 resistances');
  READLN(R1); READLN(R2); READLN(R3);
  (* On calcule X egal à 1/R *)
  X := 1/R1 + 1/R2 + 1/R3;
  (* On calcule R egal à 1/X *)
  R := 1/X;
  WRITELN('resistance equivalente = ', R)
END.
```

Exemple : Si R1 = 2, R2 = 12 et R3 = 6.38, alors la résistance équivalente est R = 1.35 .

2°- EXPRESSIONS LOGIQUES

Une variable de type logique (ou booléen) doit prendre pour valeur la constante **TRUE** (pour VRAI) ou la constante **FALSE** (pour FAUX). Ces valeurs peuvent être obtenues par **l'évaluation d'une expression logique.**

Exemple :

```
PlusGrand <--- A > B
PlusGrand := A > B
```

Selon les valeurs des entiers A et B, la variable *PlusGrand* va prendre pour valeur VRAI (TRUE) si A est supérieur à B, ou FAUX (FALSE) si A n'est pas supérieur à B. A et B peuvent être des variables numériques, caractères ou logiques (FAUX est inférieur à VRAI).

La fonction logique **ODD(*x*)**, dans laquelle *x* est un entier, donne le résultat VRAI si *x* est impair.

Une expression logique contient un ou plusieurs opérateurs relationnels et/ou un ou plusieurs opérateurs logiques.

Opérateurs relationnels	
=	égal à
<>	différent de
<	inférieur à
>	supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à

Opérateurs logiques	
NOT	(NON négation)
AND	(ET logique)
OR	(OU logique)

La priorité des opérateurs logiques est NOT puis AND puis OR. Cet ordre peut être modifié par l'utilisation de parenthèses.

Le tableau ci dessous donne les valeurs des expressions NOT A, A AND B, A OR B, pour toutes les valeurs possibles de A et de B.

Dans le tableau suivant, TRUE (VRAI) est noté T et FALSE (FAUX) est noté F. Ces valeurs seront écrites en toutes lettres dans les algorithmes et les programmes

A	B	NOT A	A AND B	A OR B
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

EXEMPLES :

- 1°
Supposons qu'un programme fasse intervenir une variable logique ***A***, et une variable ***opposedeA*** correspondant à l'opposé de ***A***. ***opposedeA*** est égal à VRAI (TRUE) si ***A*** est FAUX (FALSE), et à FAUX (FALSE) si ***A*** est VRAI (TRUE).

opposedeA := NOT A

- 2°
Pour écrire dans un programme que *j* est compris entre *i* et *k* (***i < j < k***), il faudra écrire que *i* est inférieur à *j* (***i < j***) et *j* est inférieur à *k* (***j < k***).

(i < j) AND (j < k)

- 3°
Si A = 2, B = 3 et C = 4, la valeur des variables booléennes **X, Y, Z,** calculées à partir des expressions suivantes :

X := (A < B) **AND** (C < B)
Y := (A < B) **OR** (C < B)
Z := X **OR NOT** Y

seront respectivement **FALSE, TRUE, FALSE.**

EXERCICE I-14 : Soit X,Y,Z trois variables logiques ayant respectivement pour valeurs VRAI, FAUX, VRAI. Quel est le résultat de l'évaluation des expressions logiques suivantes :

X ET Y	(FAUX)
Y OU Z	(VRAI)
NON X OU Y	(FAUX)
Y ET NON Z	(FAUX)
Z OU (NON X ET Y)	(VRAI)
NON(X OU (Y ET Z))	(FAUX)

APPLICATION I-7

Algorithme et Programme calculant et affichant la valeur des expressions de l'exercice I-14.

a) Algorithme

```
Début algorithme expressions
  logique : X = VRAI, Y = FAUX, Z = VRAI
Début action
  écrire 'Valeur de X = ',X
  écrire 'Valeur de Y = ',Y
  écrire 'Valeur de Z = ',Z
  écrire 'Valeur de X ET Y = ', X ET Y
  écrire 'Valeur de Y OU Z = ', Y OU Z
  écrire 'Valeur de NON X OU Y', (NON X) OU Y
  écrire 'Valeur de Y ET NON Z', Y ET (NON Z)
  écrire 'Valeur de Z OU (NON X ET Y)'
  écrire  Z OU ((NON X) ET Y)
  écrire 'Valeur de NON(X OU (Y ET Z)'
  écrire  NON(X OU (Y ET Z)
  Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM expressions (OUTPUT);
CONST
  X = TRUE; Y = FALSE; Z = TRUE;
BEGIN
  WRITELN('Valeur de X = ',X);
  WRITELN('Valeur de Y = ',Y);
  WRITELN('Valeur de Z = ',Z);
  WRITELN('Valeur de X ET Y = ', X AND Y);
  WRITELN('Valeur de Y OU Z = ', Y OR Z);
  WRITELN('Valeur de NON X OU Y');
  WRITELN( (NOT X) OR Y );
  WRITELN('Valeur de Y ET NON Z');
  WRITELN( Y AND (NOT Z) );
  WRITELN('Valeur de Z OU (NON X ET Y)' );
  WRITELN( Z OR ((NOT X) AND Y) );
  WRITELN('Valeur de NON(X OU (Y ET Z) )' );
  WRITELN( NOT (X OR (Y AND Z) ) )
END.
```


II – INSTRUCTIONS STRUCTUREES

4. INSTRUCTION COMPOSEE

Les Programmes précédents réalisent tous des séquences d'instructions simples. Une instruction composée est une séquence d'instructions simples apparaissant dans le corps du programme.
Les instructions simples d'une instruction composée sont toujours regroupées entre les mots BEGIN et END. Elles sont effectuées de manière séquentielle dans l'ordre où elles apparaissent dans le programme.

Algorithme	Programme Pascal

entier : x, y
Début action
lire x
y <--- 2*x +1
écrire y
Fin action
	VAR
	x, y : INTEGER;
	BEGIN
	READLN(x);
	y := 2*x +1;
	WRITE(y)
	END;

Les programmes plus élaborés, faisant notamment appel aux structures alternatives font apparaître plusieurs blocs BEGIN END. La forme générale d'un algorithme et d'un programme contenant des instructions composées est donnée ci-dessous.

Algorithme	Programme Pascal

Début action
Début action
Début action
Fin action
Fin action
Début action
Fin action
Fin action
	BEGIN
	BEGIN
	BEGIN
	END
	END;
	BEGIN
	END
	END;

5. INSTRUCTIONS ALTERNATIVES

Les instructions alternatives font intervenir une condition. Elles correspondent au test unique d'une expression logique. Le résultat va être VRAI (TRUE), si la condition est remplie et FAUX (FALSE) dans le cas contraire. Il existe 2 formes d'instructions alternatives ; l'une est dite réduite, l'autre complète.

1°- INSTRUCTION ALTERNATIVE REDUITE

L'instruction n'est exécutée que si l'évaluation de la condition ou de l'expression donne un résultat égal à VRAI (Fig.B-2).
L'instruction peut être simple ou composée. Si elle est composée, elle commence par BEGIN.

Notation algorithmique et Organigramme

```
si condition
  alors action
Fin si
```

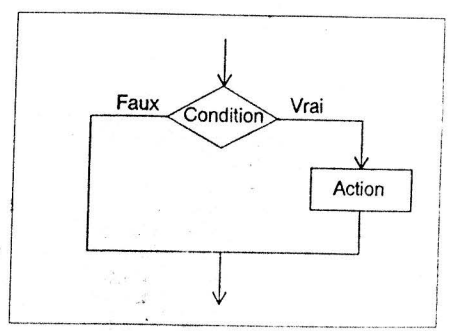


Fig. B-2. Instruction alternative réduite.

Syntaxe Pascal et Diagramme syntaxique

```
IF <expression> THEN <instruction>
```

■ EXEMPLE NUMERIQUE SIMPLE

Algorithme et Programme affichant, pour un nombre saisi par l'utilisateur, le message 'le nombre est pair' si c'est le cas.

a) Algorithme

```
Début algorithme parité
   entier : nombre
Début action
   écrire 'Saisir un nombre entier '
   lire nombre
   si nombre MOD 2 = 0      /* instruction simple */
      alors écrire 'le nombre est pair'
   Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM parite (INPUT,OUTPUT);
VAR
   nombre : INTEGER;
BEGIN
   WRITE('Saisir un nombre entier ');
   READLN(nombre);
   IF nombre MOD 2 = 0     (* instruction simple *)
      THEN WRITELN('le nombre est pair')
END.
```

Remarque 1 : Cet exemple utilise la fonction MOD pour tester la parité du nombre. Rappel : Il n'y a jamais de point virgule devant un END.

Remarque 2 : Les instructions sont décalées en respectant les différents niveaux auxquels elles appartiennent. Le respect de ces décalages s'appelle l'**Indentation** ; elle est impérative pour une bonne lisibilité du programme.

APPLICATION II-1

Algorithme et Programme calculant la prime d'ancienneté d'un salarié, sachant qu'elle représente 50 F par année d'ancienneté. Une saisie d'ancienneté négative provoque l'affichage d'un message d'erreur.

a) Algorithme

```
Début algorithme prime
   entier : ancien, prime
   logique : saisiok
Début action
   saisiok <--- VRAI
   écrire 'Saisir l"ancienneté '
   lire ancien
   si ancien < 0
      alors                  /* instruction composée */
         écrire 'Erreur de saisie'
         saisiok <--- FAUX
   Fin si
   si saisiok
      alors                  /* instruction composée */
         prime <--- ancien * 50
         écrire 'Prime ancien = ', prime : 4
   Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM prime (INPUT,OUTPUT);
VAR
   ancien, prime : INTEGER;
   saisiok : BOOLEAN;
BEGIN
   saisiok := TRUE;
   WRITE('Saisir l"anciennete ');
   READLN(ancien);
   IF ancien < 0
      THEN                   (* instruction composee *)
         BEGIN
            WRITELN('Erreur de saisie');
            saisiok := FALSE
         END;
   IF saisiok
      THEN                   (* instruction composee *)
         BEGIN
            prime := ancien * 50;
            WRITELN('Prime ancien = ', prime : 4)
         END
END.
```

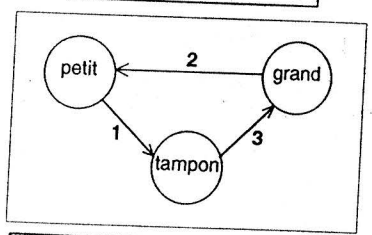

APPLICATION II-2

Ecrire un Algorithme, sa trace et le programme qui ordonne dans le sens croissant 2 variables en échangeant leurs valeurs si nécessaire.

a) Algorithme

```
Début algorithme ordonner
   réel : petit, grand, tampon
   logique : ordrecorrect
Début action
   écrire 'Saisir la 1ère valeur '
   lire petit
   écrire 'Saisir la 2ème valeur '
   lire grand
   ordrecorrect <--- petit < grand
   si NON ordrecorrect
      alors
         tampon <--- petit
         petit <--- grand
         grand <--- tampon
   Fin si
   écrire 'Valeurs ordonnées : '
   écrire petit :4:2,' ',grand :4:2
Fin action
Fin algorithme
```

b) Trace de l'échange



Etape	tampon	petit	grand
		7	4
1	7	7	4
2	7	4	4
3	7	4	7

Remarque 1 : Si l'ordre n'est pas correct, l'échange des valeurs utilise une variable tampon pour sauvegarder la valeur de la variable *petit*.
Remarque 2 : Plusieurs instructions simples sont nécessaires pour effectuer l'échange. Elles sont regroupées dans le programme entre le couple <BEGIN END> d'une instruction composée.

c) Programme Pascal

```
PROGRAM ordonner (INPUT,OUTPUT);
VAR   petit, grand, tampon : REAL;
      ordrecorrect : BOOLEAN;
BEGIN
   WRITE('Saisir la 1ere valeur ');
   READLN(petit);
   WRITE('Saisir la 2eme valeur ');
   READLN(grand);
   ordrecorrect := petit < grand;
   IF NOT ordrecorrect
      THEN
      BEGIN
         tampon := petit;
         petit := grand;
         grand := tampon
      END;
   WRITELN('Valeurs ordonnees : ');
   WRITELN(petit :4:2,' ',grand :4:2)
END.
```

2°- INSTRUCTION ALTERNATIVE COMPLETE

Elle exécute une instruction (simple ou composée) si l'évaluation de l'expression (Fig.B3) donne un résultat égal à VRAI (TRUE) ou une autre instruction si le résultat est FAUX (FALSE).

Notation algorithmique et Organigramme

```
si condition
   alors action1
   sinon action2
Fin si
```

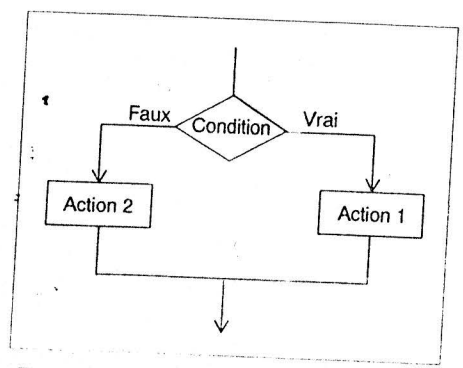
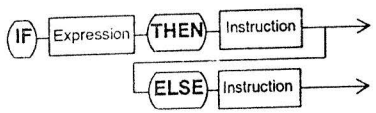


Fig. B-3. Instruction alternative complète.

Syntaxe Pascal et Diagramme syntaxique

```
IF <expression>
   THEN <instruction1>
   ELSE <instruction2>
```



■ EXEMPLE NUMERIQUE SIMPLE

Algorithme et Programme de **Triage d'une pièce en fonction de son poids et d'un paramètre de poids maximum**, saisis par l'Utilisateur.

a) Algorithme

```
Début algorithme triage
  réel : poids, parametre
Début action
  écrire 'Saisir le paramètre poids maximum '
  lire parametre
  écrire 'Lecture du poids de la pièce'
  lire poids
  si poids > parametre
    alors écrire 'oriente vers magasin A '
    sinon écrire 'oriente vers magasin B '
  Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM triage (INPUT, OUTPUT);
VAR
  poids, parametre : REAL;
BEGIN
  WRITE('Saisir le parametre poids maximum ');
  READLN(parametre);
  WRITE('Lecture du poids de la piece');
  READLN(poids);
  IF poids > parametre
    THEN WRITELN('oriente vers magasin A ')
    ELSE WRITELN('oriente vers magasin B ')
END.
```

Remarque : Il n'y a jamais de point virgule devant un ELSE.

APPLICATION II-3

En fonction de la reponse à une question, **convertir des degrés en radians ou des radians en degrés.** Algorithme et Programme.

a) Algorithme

```
Début algorithme conversion
  réel : pi = 3.14, deg, rad
  caractère : choix
Début action
  écrire 'd = deg en rad,  ou r = rad en deg ? '
  lire choix
  si choix = 'd'
    alors
      écrire 'Saisir une valeur en degré '
      lire deg
      rad := deg * pi / 180
      écrire 'valeur en radian =  ', rad
    sinon
      écrire 'saisir une valeur en radian '
      lire rad
      deg := rad * 180 / pi;
      écrire 'valeur en degré =  ', deg
  Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM conversion (INPUT,OUTPUT);
CONST   pi = 3.14;
VAR     deg,rad : REAL;
        choix : CHAR;
BEGIN
  WRITE('d = deg en rad, ou r = rad en deg ? ');
  READLN(choix);
  IF choix = 'd'
    THEN
      BEGIN
        WRITE('Saisir une valeur en degre ');
        READLN(deg);
        rad := deg * pi / 180;
        WRITELN ('valeur en radian =  ', rad)
      END
    ELSE
      BEGIN
        WRITE ('saisir une valeur en radian ');
        READLN (rad);
        deg := rad * 180 / pi;
        WRITELN ('valeur en degre =  ', deg)
      END
END.
```


APPLICATION II-4

Débiter un stock de 50 pièces ou le réapprovisionner par lot de 100 unités si la quantité commandée, comprise entre 1 et 100, est supérieure à la quantité en stock (toujours initialisée à 50).

a) Algorithme

```
Début algorithme stock1
  entier : qtestock, qtecomm
Début action
  qtestock <--- 50
  écrire 'Saisir quantité commandée (1 à 100) '
  lire qtecomm
  si qtecomm <= qtestock
    alors
      qtestock <--- qtestock - qtecomm
      écrire 'Quantité sortie ',qtecomm :3
    sinon
      écrire 'Sortie impossible'
      écrire 'Réapprovisionnement du Stock'
      qtestock <--- qtestock + 100
  Fin si
  écrire 'Nouveau stock ',qtestock :3
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM stock1 (INPUT,OUTPUT);
VAR
  qtestock, qtecomm : INTEGER;
BEGIN
  qtestock := 50;
  WRITE('Saisir quantite commandee (1 à 100)');
  READLN(qtecomm);
  IF qtecomm <= qtestock
    THEN
    BEGIN
      qtestock := qtestock - qtecomm;
      WRITELN('Quantite sortie ',qtecomm :3)
    END
    ELSE
    BEGIN
      WRITELN('Sortie impossible');
      WRITELN('Reapprovisionnement Stock');
      qtestock := qtestock + 100
    END;
  WRITELN('Nouveau stock ',qtestock :3)
END.
```

Remarque : Les instructions suivant le THEN et le ELSE sont des instructions composées, incluses dans des couples BEGIN END.
L'instruction commençant par IF se termine ici par le END précédant l'instruction WRITELN.

EXERCICE II-1 : Après saisie d'un prix à payer, et de la somme versée par le client, afficher la somme à rendre si elle est positive.

EXERCICE II-2 : Calculer la remise sur l'achat d'un véhicule sachant que le taux de remise est de 5 % sur un prix inférieur à 70 000 F et de 6 % sur un prix supérieur à 70 000 F.

3°- UTILISATION DE PLUSIEURS —— OPERATEURS LOGIQUES ——

Les applications et exemples donnés sur les instructions alternatives réduite et complète utilisent tous une expression simple de la condition testée. Le test porte sur une expression mettant en oeuvre un opérateur numérique ou un opérateur logique :

```
si nombre MOD 2 = 0 alors ...          (égal à)
si ancien < 0 alors ...                (inférieur à)
si saisiok alors ...                   (variable logique)
si NON ordrecorrect alors ...          (opérateur logique)
si poids > parametre alors ...         (supérieur à)
si choix = 'd' alors ...               (égal à)
si qtecomm < qtestock alors ...        (inférieur à)
```

Les expressions peuvent aussi être composées de plusieurs facteurs faisant intervenir des opérateurs numériques et logiques.

Les exemples suivants montrent des expressions composées de deux facteurs, séparés par un opérateur logique.

```
si (nombre MOD 2 = 0) et (nombre > 100) alors ...
si (ancien < 0) ou saisiok alors ...
si (poids > parametre) et (choix = 'd') alors ...
```

Traduction en Pascal

```
IF (nombre MOD 2 = 0) AND (nombre > 100)
  THEN . .
IF (ancien < 0) OR saisiok
  THEN ...
IF (poids > parametre) AND (choix = 'd')
  THEN ...
```

APPLICATION II-5

Dans le circuit électrique illustré par la figure B-4, la lampe s'allume si le contact C est Fermé OU si les contacts A ET B sont Fermés.
On demande à l'utilisateur de saisir l'état ('F' = Fermé ou 'O' = Ouvert, 'F' et 'O' en Majuscules) de chacun des 3 contacts A, B et C. En fonction de cette saisie, afficher le message *'Lampe allumee'* ou le message *'Lampe eteinte'*.

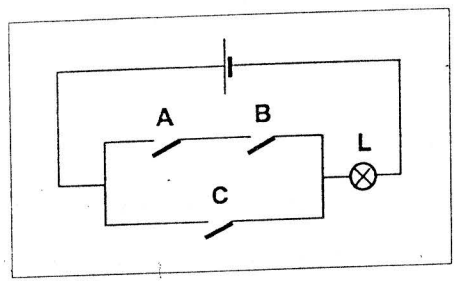


Fig. B-4. Allumage d'une Lampe.

a) Algorithme

```
Début Algorithme allumage1
   caractère : A, B, C
Début action
   écrire 'F = Fermé, O = Ouvert, Majuscule'
   écrire 'Saisir valeurs des contacts A, B, C '
   écrire 'exemple : FOF et taper sur ENTREE'
   lire A, B, C
   si C = 'F' OU (A = 'F' et B = 'F')
      alors écrire 'Lampe allumée'
      sinon écrire 'Lampe éteinte'
   Fin si
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM allumage1 (INPUT,OUTPUT);
VAR
   A, B, C : CHAR;
BEGIN
   WRITELN('F = Ferme, O = Ouvert, Majuscule');
   WRITELN('Saisir valeurs des contacts A,B,C');
   WRITELN('exemple : FOF et taper ENTREE');
   READLN(A, B, C);
   IF (C = 'F') OR ((A = 'F') AND (B = 'F'))
      THEN WRITELN('Lampe allumee')
      ELSE WRITELN('Lampe eteinte')
END.
```

4°- INSTRUCTIONS ALTERNATIVES IMBRIQUEES

L'exercice ci-contre est résolu en testant une expression composée de plusieurs facteurs réunis par des opérateurs logiques.

Le contact C est testé. SI C = 'F' ALORS le message 'Lampe allumee' est affiché, SINON on teste A et B.

Cette structure de tests nous sert de base pour illustrer les instructions alternatives imbriquées.

a) Algorithme

```
Début Algorithme allumage2
   caractère : A, B, C
Début action
   écrire 'F = Fermé, O = Ouvert, Majuscule'
   écrire 'Saisir valeurs des contacts A, B, C '
   écrire 'exemple : FOF et taper sur ENTREE'
   lire A, B, C
   si C = 'F'
      alors écrire 'Lampe allumée'
      sinon
         si A = 'F' et B = 'F'
            alors écrire 'Lampe allumée'
            sinon écrire 'Lampe éteinte'
         Fin si
   Fin si
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM allumage2 (INPUT,OUTPUT);
VAR   A, B, C : CHAR;
BEGIN
   WRITELN('F = Ferme, O = Ouvert, Majuscule');
   WRITELN('Saisir valeurs des contacts A,B,C');
   WRITELN('exemple : FOF et taper ENTREE');
   READLN(A, B, C);
   IF C = 'F'
      THEN WRITELN('Lampe allumee')
      ELSE
         IF (A = 'F') AND (B = 'F')
            THEN WRITELN('Lampe allumee')
            ELSE WRITELN('Lampe eteinte')
END.
```

Remarque : Les programmes allumage1 et allumage2 sont rigoureusement équivalents. Ils ne testent A et B que si la valeur de C est différente de 'F'.

6. INSTRUCTION DE CONDITIONS MULTIPLES

L'instruction de Condition Multiple utilise une variable ou expression de référence, appelée **sélecteur**, dont la valeur est comparée à une liste d'étiquettes. En cas d'égalité avec une étiquette, l'instruction associée à l'étiquette est exécutée.

Notation algorithmique

```
selon que sélecteur
   étiquette1 : action1
   étiquette2 : action2
   ...
   etiquetten : actionn
Fin selon
```

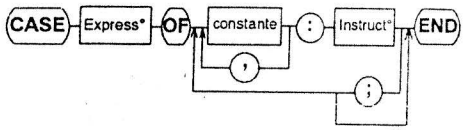
Cette instruction est comparable mais plus concise et plus claire qu'une série de tests portant sur différentes valeurs **d'une même variable** ou expression, comme s'il s'agissait d' instructions <si alors sinon> imbriquées dans lesquelles une variable *selecteur* serait testée sur sa valeur :

```
si selecteur = valeur1
   alors <action1>
   sinon
      si selecteur = valeur2
         alors <action2>
         sinon
            si selecteur = valeur3
               alors <action3>
               sinon
                  ...
               Fin si
         Fin si
Fin si
```

Si la 1ère condition n'est pas remplie, le résultat du test est FAUX, et le programme passe à la condition suivante. Si une condition est remplie, le résultat du test est VRAI, l'action correspondante est exécutée, et on sort de la structure.

Syntaxe Pascal et Diagramme syntaxique

```
CASE <expression> OF
 <etiquette1> : <instruction1>
 <etiquette2> : <instruction2>
 ...
 <etiquettenn> : <instructionn>
END;
```



■ EXEMPLE NUMERIQUE SIMPLE

Affectation de la durée d'une tâche, en fonction du numéro d'étape entrant dans un processus de traitement. Les tâches N°1 et N°4 ont une durée de 15 secondes, les tâches N°2 et N°3 durent respectivement 10 et 25 secondes, l'étape N°5, de durée 0, déclenche l'arrêt du processus.

a) Algorithme

```
Début algorithme processus
   entier : etape, duree
   logique : arret
Début action
   arret <--- FAUX, duree <--- 0
   écrire 'Lecture Numéro Etape '
   lire etape
   selon que etape
      1 : duree <--- 15
      2 : duree <--- 10
      3 : duree <--- 25
      4 : duree <--- 15
      5 : arret <--- VRAI
   Fin selon
   écrire duree, '   ', arret
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM processus (INPUT,OUTPUT);
VAR    etape, duree : INTEGER;
       arret : BOOLEAN;
BEGIN
   arret := FALSE; duree := 0;
   WRITE('Lecture Numero Etape ');
   READLN(etape);
   CASE etape OF
      1 : duree := 15;
      2 : duree := 10;
      3 : duree := 25;
      4 : duree := 15;
      5 : arret := TRUE
   END;
   WRITELN(duree, '   ', arret)
END.
```

APPLICATION II-6

Une compagnie d'assurances calcule le montant de la prime pour un véhicule à partir d'une BASE égale à 1000 F, et des conditions :

Risque :	Tierce Collision	Prime = Base
	Tout Risque	Prime = Base * 1.5
Puissance :	2 à 5 CV	Coefficient de 1.00
	6 à 8 CV	Coefficient de 1.20
	9 CV et plus	Coefficient de 1.40
Utilisation :	Promenade	Coefficient de 0.90
	Trajet	Coefficient de 1.10
	Affaire	Coefficient de 1.25

Etablir l'Algorithme et le programme de Calcul des primes d'assurance.

a) Algorithme

```
Début algorithme assurance
  caractère : risque, utilisation
  entier : puissance, base= 1000
  réel : prime
Début action
  /* Saisie des données */
  écrire 'Saisir le Risque (Majuscule)'
  écrire 'C = Collision, T = Tout risque'
  lire risque
  écrire 'Saisir la Puissance du véhicule'
  lire puissance
  écrire 'Saisir Type d"utilisation'
  écrire 'P = Promenade, T = Trajet, A = Affaire'
  lire utilisation
  /* Calcul de la prime */
  si risque = 'C'
    alors prime <--- base
    sinon prime <--- base * 1.5
  Fin si
  si puissance >= 9
    alors prime <--- prime * 1.4
    sinon si puissance > 5
          alors prime <--- prime * 1.2
          sinon prime <--- prime * 1.0
        Fin si
  Fin si
  selon que utilisation
    'P' : prime <--- prime * 0.90
    'T' : prime <--- prime * 1.10
    'A' : prime <--- prime * 1.25
  Fin selon
  écrire 'Montant de la prime = ', prime :6:2
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM assurance (INPUT, OUTPUT);
CONST
  base = 1000;
VAR
  puissance : INTEGER;
  risque, utilisation : CHAR;
  prime : REAL;

BEGIN
  (* Saisie des donnees *)
  WRITELN('Saisir le Risque (Majuscule)');
  WRITELN('C = Collision, T = Tout risque');
  READLN(risque);
  WRITELN('Saisir la Puissance du vehicule');
  READLN(puissance);
  WRITELN('Saisir Type d"utilisation');
  WRITELN('P=Promenade,T=Trajet,A=Affaire');
  READLN(utilisation);

  (* Calcul de la prime *)
  IF risque = 'C'
    THEN prime := base
    ELSE prime := base * 1.5;
  IF puissance >= 9
    THEN prime := prime * 1.4
    ELSE
      IF puissance > 5
        THEN prime := prime * 1.2
        ELSE prime := prime * 1.0;
  CASE utilisation OF
    'P' : prime := prime * 0.90;
    'T' : prime := prime * 1.10;
    'A' : prime := prime * 1.25
  END;
  WRITELN('Montant de la prime = ', prime :6:2)
END.
```

Remarque : Le premier test (instruction alternative complète) est effectué sur la variable *risque*, puis s'enchaînant en séquence, le second test (instructions alternatives imbriquées) porte sur la variable *puissance*. Enfin, la variable *utilisation* est testée à l'aide d'une instruction de condition multiple. Le sélecteur est la variable *utilisation*, de type caractère.

IMPORTANT

Le sélecteur peut être une expression de sélection ; mais dans tous les cas, la valeur obtenue **doit être de type caractère, entier, ou autre type scalaire.**
Le sélecteur ne peut pas être de type réel.

APPLICATION II-7

Une Entreprise pratique les conditions de vente suivantes, en fonction du conditionnement du produit :
- Prix de base pour vente à l'unité (U).
- Réduction de b% sur le prix de base pour vente par boite de 10 (B).
- Prix de gros pour vente par carton de 100 (C).
- Réduction de g% sur le prix de gros pour vente par palette de 1000 (P).

Sachant qu'un taux de TVA de 18,6% s'applique sur ce produit, calculer le prix TTC d'une commande pour une quantité commandée dans un type de conditionnement.

a) Algorithme

```
Début algorithme prix
  réel : tva = 0.186, reducbase = 0.05,
         reducgros = 0.15,
         pribase, prigros, priht, totalht, totalttc
  caractère : conditmt   /* Valeurs 'U','B','C','P' */
              /* Unité, Boite, Carton et Palette */
  entier : qtecomm, nbunite
           /* quantité commandée et nombre */
           /* d'unité dans le conditionnement */
Début action
  /* Saisie des données */
  écrire 'Saisir prix de base du produit'
  lire pribase
  écrire 'Saisir prix de gros du produit'
  lire prigros
  écrire 'Saisir Type de conditionnement '
  écrire 'U = Unité, B = Boite, '
  écrire 'C = Carton, P = Palette'
  lire conditmt
  écrire 'Saisir la quantité commandée'
  lire qtecomm
  selon que conditmt          /* Calcul du prix */
    'U' : priht <--- pribase
          nbunite <--- 1
    'B' : priht <--- pribase * (1 - reducbase)
          nbunite <--- 10
    'C' : priht <--- prigros
          nbunite <--- 100
    'P' : priht <--- prigros * (1 - reducgros)
          nbunite <--- 1000
  Fin selon
  totalht <--- qtecomm * nbunite * priht
  totalttc <--- totalht * (1 + tva)
  /* affichage du résultat */
  écrire 'Prix TTC commande ', totalttc :6:2
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM prix (INPUT, OUTPUT);
CONST  tva = 0.186;
       reducbase = 0.05; reducgros = 0.15;
VAR
  pribase, prigros, priht, totalht, totalttc : REAL;
  conditmt : CHAR;      (* Valeurs 'U','B','C','P' *)
                        (* Unite, Boite, Carton, Palette *)
  qtecomm : INTEGER;  (* qtite commandee *)
  nbunite : INTEGER;  (* nombre d'unite dans *)
                      (* le conditionnement *)
BEGIN
  (* Saisie des donnees *)
  WRITELN('Saisir prix de base du produit');
  READLN(pribase);
  WRITELN('Saisir prix de gros du produit');
  READLN(prigros);
  WRITELN('Saisir Type de conditionnement ');
  WRITE('U = Unite, B = Boite, ');
  WRITELN('C = Carton, P = Palette');
  READLN(conditmt);
  WRITELN('Saisir la quantite commandee');
  READLN(qtecomm);
  CASE conditmt OF      (* Calcul du prix *)
    'U' : BEGIN
            priht := pribase;
            nbunite := 1
          END;
    'B' : BEGIN
            priht := pribase * (1 - reducbase);
            nbunite := 10
          END;
    'C' : BEGIN
            priht := prigros;
            nbunite := 100
          END;
    'P' : BEGIN
            priht := prigros * (1 - reducgros);
            nbunite := 1000
          END
  END;
  totalht := qtecomm * nbunite * priht;
  totalttc := totalht * (1 + tva);
  (* affichage du resultat *)
  WRITELN('Prix TTC commande ', totalttc :6:2)
END.
```

Remarque : Chaque étiquette correspond ici à une instruction composée (couple BEGIN .. END).

EXERCICE II-3 : Calculer le salaire brut d'un Ouvrier Qualifié, en fonction du nombre d'heures travaillées et de son niveau : Niveau 1 à 37 F/H, Niveau 2 à 39,20 F/H, Niveau 3 à 41,80 F/H, Niveau 4 à 44,60 F/H.

III – INSTRUCTIONS REPETITIVES

7. TRAITEMENT D'UNE FILE

■ NOTION DE FILE

Considérons une Machine qui génére de manière continue un certain nombre d'éléments de même nature. Par exemple, une Presse produit une série finie de x pièces du même type.
Supposons que chaque pièce produite chute sur une bande transporteuse qui va la convoyer vers un dispositif automatique de contrôle opérant un certain nombre d'opérations (mesures, comparaison avec un standard, rejet de la pièce non conforme au standard, comptage ...).
Le dispositif de contrôle réalise sur chaque pièce le même algorithme de traitement.

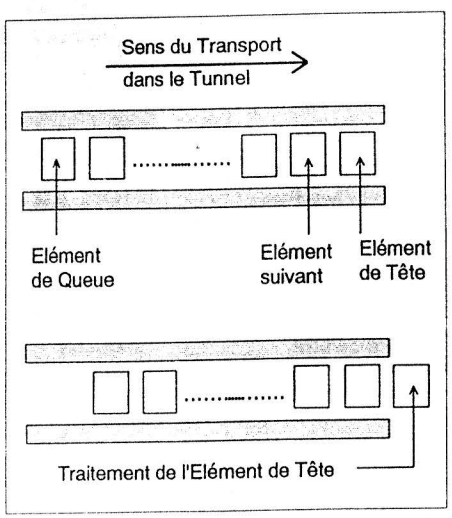


Fig. B–5. Représentation d'une File

Une file peut être définie comme une série finie de données relatives à un même objet sur lequel s'applique un traitement répétitif.

La figure B–5 représente une file sous la forme d'un tunnel dans lequel les éléments progressent. Le traitement est effectué à la sortie du tunnel.
Les instructions répétitives (ou itératives) de l'algorithmique et du Pascal sont spécifiques au traitement des différents types de File.

■ TYPES DE FILE ET TRAITEMENTS

1. Le nombre d'éléments contenus dans la file peut être connu. Dans ce cas, on sait de manière exacte combien de fois le traitement devra être effectué et on utilisera la structure POUR (**FOR**).

Si le nombre d'éléments de la file est inconnu, il faut s'interroger sur l'élément de queue, c'est à dire le reconnaître et décider s'il doit ou non déclencher un ultime traitement.

2. L'élément de queue peut être un repère de nature différente des éléments précédents de la file, on décidera alors de ne pas le traiter en utilisant la structure TANT QUE (**WHILE**).

3. L'élément de queue peut au contraire être un repère de même nature que ses prédécesseurs, et on décidera de le traiter en utilisant la structure REPETER (**REPEAT**).

8. STRUCTURE POUR...

Lorsque le nombre d'éléments de la file est connu, on peut envisager 2 modes de traitement.
Le traitement des éléments est possible :
– soit à partir du premier (sens croissant),
– soit à partir du dernier (sens décroissant).
Pour illustrer ce dernier cas, il suffit d'inverser le sens du transport de la figure B–5, et de commencer le traitement par l'élément de queue, qui devient alors la tête.
Le traitement peut aussi n'être effectué que pour 1 élément sur n (n s'appelle le pas ou l'incrément).

Notation algorithmique Sens Croissant

```
Pour indice de Début à Fin par pas de n faire
   action
Fin Pour
```

Lorsque n = 1, la mention par pas de 1 est inutile.
Les valeurs de Début, Fin et pas sont des entiers.
La variable entière *indice* prendra les valeurs Début, Début + n, Début + n + n ..., jusqu'à ce qu'elle devienne supérieure à la valeur Fin.

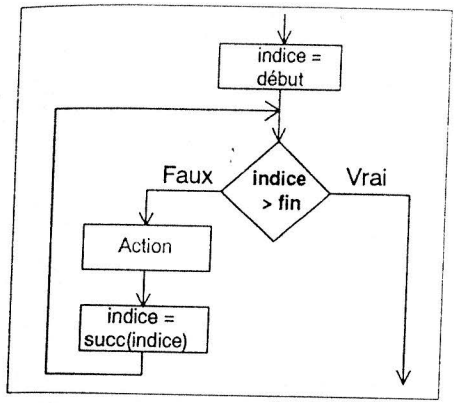


Fig. B–6. Structure POUR Croissante

Notation algorithmique Sens Décroissant

```
Pour indice de Fin à Début par pas de n faire
   action
Fin Pour
```

La variable entière *indice* prendra les valeurs Fin, Fin – n, Fin – n – n ..., jusqu'à ce qu'elle devienne inférieure à la valeur Début.

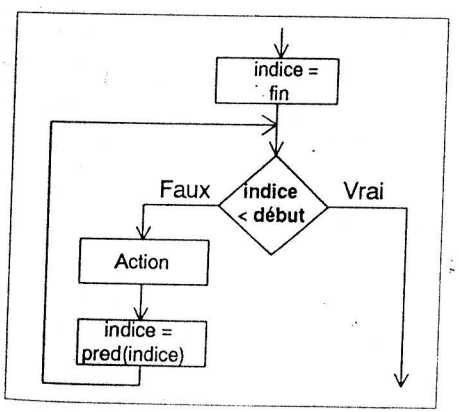


Fig. B–7. Structure POUR Décroissante

L'action est effectuée pour chacune des valeurs valides de l'indice. L'incrémentation est effectuée automatiquement dans la structure. Le programmeur ne doit pas intervenir sur l'indice dans une structure Pour.

Syntaxe Pascal * Sens croissant *

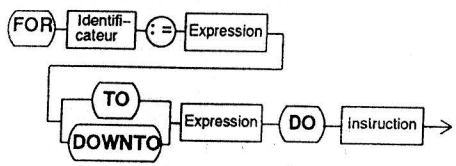
```
FOR indice := Début TO Fin DO <instruction>
```

Syntaxe Pascal * Sens décroissant *

```
FOR indice := Fin DOWNTO Début DO <instruction>
```

Le Pascal standard ne dispose pas d'instruction équivalente à la notation algorithmique par pas de.

Diagramme syntaxique du FOR



■ EXEMPLE NUMERIQUE SIMPLE

Algorithme (avec pas) et Programme (avec un test) calculant la somme des nombres pairs sur la série des 10 premiers nombres entiers.

a) Algorithme

```
Début algorithme somme1
   entier : i, som
Début action
   écrire 'Somme nombres pairs de 1 a 10'
   som <--- 0
   Pour i de 2 à 10 par pas de 2 faire
      som <--- som + i
   Fin pour
   écrire 'somme = ', som : 2
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM somme1 (OUTPUT);
VAR   i, som ; INTEGER;
BEGIN
   WRITELN('Somme nombres pairs de 1 a 10');
   som := 0;
   FOR I := 2 TO 10 DO
      IF I MOD 2 = 0 THEN som := som + I;
   WRITELN('somme = ', som : 2)
END.
```

APPLICATION III-1

Recherche du minimum et du maximum dans une série de n mesures (n saisi positif par l'utilisateur). Les mesures ont des valeurs entières positives.

a) Algorithme

```
Début algorithme minmax
  entier : min, max, n, i, mesure
Début action
  /* on donne au minimum la valeur la plus
  grande possible, et au maximum la valeur la
  plus petite possible */
  min <--- 32767              /* entier maximum */
  max <--- 0
  écrire 'Saisir le nombre de mesures'
  lire n
  Pour I de 1 à n faire
    écrire 'Saisir la mesure numéro ', i
    lire mesure
    si mesure < min
      alors min <--- mesure
      sinon
        si mesure > max
          alors max <--- mesure
        Fin si
    Fin si
  Fin pour
  écrire 'Minimum des mesures ', min
  écrire 'Maximum des mesures ', max
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM minmax (INPUT,OUTPUT);
VAR   min, max, n, i, mesure : INTEGER;
BEGIN
  max := 0;  min := MAXINT; (* entier maximum *)
  WRITELN('Saisir le nombre de mesures');
  READLN(n);
  FOR I := 1 TO n DO
    BEGIN
    WRITELN('Saisir la mesure numero ', i);
    READLN(mesure);
    IF mesure < min
      THEN min := mesure
      ELSE
        IF mesure > max THEN max := mesure
    END;
  WRITELN('Minimum des mesures ', min);
  WRITELN('Maximum des mesures ', max)
END.
```

APPLICATION III-2

Contrôle pondéral d'une série de 50 pièces. On calculera le poids moyen des pièces de cet ensemble ainsi que les nombres et poids moyens des pièces dont le poids dépasse les limites inférieure et supérieure, paramètrées par l'utilisateur. Etablir un algorithme, le programme correspondant et expliquer le programme ligne par ligne.

a) Algorithme

```
Début algorithme controle de poids
  entier : i, ninf, nsup
  /* i = indice, ninf et nsup = nombres de pièces
  de poids inférieur et supérieur aux limites */
  réel :p,                    /* poids d'une pièce */
       liminf, limsup, /* limites inf. et supérieure */
       pt50, ptinf, ptsup,    /* poids totaux */
       pm50, pminf, pmsup     /* poids moyens */
Début action
  écrire 'Saisir la limite de poids inférieure'
  lire liminf
  écrire 'Saisir la limite de poids supérieure'
  lire limsup
  ninf <--- 0, nsup <--- 0
  ptinf <--- 0, ptsup <--- 0
  pt50 <--- 0
  Pour I de 1 à 50 faire
    écrire 'Mesure du poids de la pièce'
    lire p
    pt50 := pt50 + p
    si p < liminf
    alors
      ninf <--- ninf + 1
      ptinf <--- ptinf + p
    sinon
      si p > limsup
      alors
        nsup <--- nsup + 1
        ptsup <--- ptsup + p
      Fin si
    Fin si
  Fin pour
  écrire 'poids moyen des 50 pièces', pt50 / 50
  si ninf <> 0 alors
    écrire 'nombre pièces de poids faible', ninf
    écrire 'poids moyen inférieur', ptinf/ninf
  Fin si
  si nsup <> 0 alors
    écrire 'nombre pièces de poids fort', nsup
    écrire 'poids moyen supérieur', ptsup / nsup
  Fin si
Fin action
Fin algorithme
```


b) Programme Pascal

```
PROGRAM controlepoids (INPUT, OUTPUT);

VAR
  i, ninf, nsup : INTEGER;
  p, liminf, limsup : REAL;
  pt50, ptinf, ptsup : REAL;
  pm50, pminf, pmsup : REAL;

BEGIN
  WRITELN('Saisir la limite de poids inferieure');
  READLN(liminf);
  WRITELN('Saisir la limite de poids superieure');
  READLN(limsup);
  ninf := 0; nsup := 0;
  ptinf := 0; ptsup := 0; pt50 := 0;

  FOR I := 1 TO 50 DO
  BEGIN
    WRITELN('Mesure du poids de la piece');
    READLN(p);
    pt50 := pt50 + p;
    IF p < liminf
    THEN
      BEGIN
        ninf := ninf + 1;  ptinf := ptinf + p
      END
    ELSE
      IF p > limsup
      THEN
        BEGIN
          nsup := nsup + 1;  ptsup := ptsup + p
        END
  END;

  WRITELN('poids moyen 50 pieces', pt50 /50);
  IF ninf <> 0
  THEN
    BEGIN
    WRITELN('nombre pieces poids faibl ', ninf);
    WRITELN('poids moyen infer. ', ptinf / ninf)
    END;
  IF nsup <> 0
  THEN
    BEGIN
    WRITELN('nombre pieces poids fort ', nsup);
    WRITELN('poids moyen super. ',ptsup/nsup)
    END
END.
```

EXERCICE III-1 : Calculer et afficher la moyenne de N notes obtenues par un élève. La valeur du nombre entier N, puis les notes, seront saisies par l'utilisateur.

c) Explications ligne par ligne

EN-TETE DE PROGRAMME

PARTIE DE DECLARATION DES VARIABLES
Ligne déclarant les variables de type **ENTIER**
Une ou Plusieurs lignes
déclarant les variables de type **REEL**

DEBUT DE L'ACTION PRINCIPALE

Lecture de la limite inférieure de poids

Lecture de la limite supérieure de poids

Initialisations : Les compteurs sont mis à zéro

DEBUT DE LA BOUCLE FOR
pour 50 itérations (de 1 à 50)

Lecture du poids de la pièce
Calcul du poids total par cumul
SI le poids est inférieur à la limite inférieure
ALORS
Comptage de la pièce par incrémentation
Calcul du poids total inférieur par cumul

SINON
SI poids supérieur à la limite supérieure
ALORS
Comptage pièce par incrémentation
Calcul poids total supérieur par cumul

FIN DE LA BOUCLE FOR

Ecriture du poids moyen des pièces
SI nombre de pièces de poids faible NON NUL
ALORS
(Ce test est impératif pour éviter une éventuelle division par ZERO)
Ecriture du nombre de pièces de poids faible
Ecriture poids moyen des pièces poids faible
SI nombre pièces de poids fort NON NUL
ALORS
(Ce test est impératif pour éviter une éventuelle division par ZERO)
Ecriture du nombre de pièces de poids fort
Ecriture poids moyen des pièces poids fort
FIN D'ACTION PRINCIPALE ET PROGRAMME

EXERCICE III-2 : Calculer et afficher année par année, le montant du capital et des intérêts acquis, pour une somme de S francs, placée à P pour cent pendant A Années.

STRUCTURE TANT QUE...

Lorsque le nombre d'éléments d'une file est inconnu, l'élément de queue peut ne pas être traité. Autrement dit, le traitement est effectué **Tant que la condition correspondant à <NON élément de queue> est égale à VRAI.**

Notation algorithmique et Organigramme

```
Tant que condition faire
    action
Fin Tant que
```

La condition est d'abord testée, puis si elle est satisfaite, l'action est exécutée.
Le test précède toujours l'action.

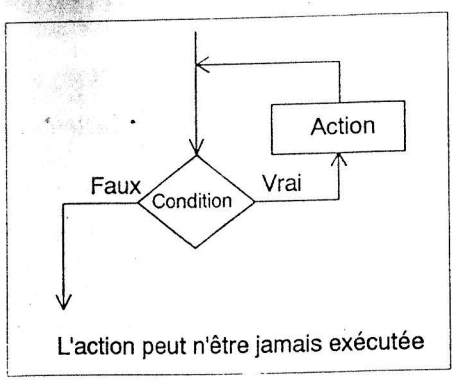
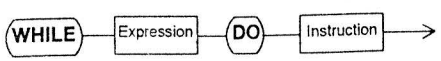


Fig. B-8. Structure TANT QUE...

Syntaxe du WHILE et Diagramme syntaxique

WHILE <expression logique> **DO** <instruction>



L'expression est de type booléen ; son évaluation fournit le résultat TRUE (VRAI) ou FALSE (FAUX). Si le résultat est TRUE, l'instruction est exécutée, puis l'expression est évaluée à nouveau. Quand le résultat est FALSE, on sort de la boucle sans exécuter l'instruction.

Remarque 1 : L'instruction peut n'être jamais exécutée dans la mesure ou dès la première évaluation de l'expression, un résultat FALSE (FAUX) peut être obtenu.

Remarque 2 : Il faut être très vigilant au fait que l'instruction doit pouvoir modifier l'expression, qui doit fournir un résultat FALSE (FAUX) après un certain nombre d'itérations. **Dans le cas contraire, la boucle serait infinie.**

■ EXEMPLE NUMERIQUE SIMPLE

Algorithme et Programme calculant la somme d'une série de nombres entiers positifs saisis par l'utilisateur. **La saisie est terminée par un nombre négatif, élément de queue de la file, qui ne sera donc pas traité.**

a) Algorithme

```
Début algorithme somme2
   entier : nombre, som
Début action
   som <--- 0
   écrire 'Saisir un nombre entier '
   lire nombre
   Tant que nombre >= 0 faire
      som <--- som + nombre
      écrire 'Saisir nombre (<0 pour Fin) '
      lire nombre
   Fin Tant que
   écrire 'somme = ', som : 5
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM somme2 (INPUT,OUTPUT);
VAR  nombre, som : INTEGER;
BEGIN
   som := 0;
   WRITE('Saisir un nombre entier ');
   READLN(nombre);
   WHILE nombre >= 0 DO
      BEGIN
         som := som + nombre;
         WRITE('Saisir nombre (<0 pour Fin) ');
         READLN(nombre)
      END;
   WRITELN('somme = ', som : 5)
END.
```


APPLICATION III-3

L'utilisateur saisit une durée comprise et contrôlée entre 5000 et 30000 unités de temps. Dès que la saisie est correcte, la vanne s'ouvre (**Fig. B-9**) et ne se ferme qu'après la sortie d'une boucle d'attente correspondant à la durée saisie.

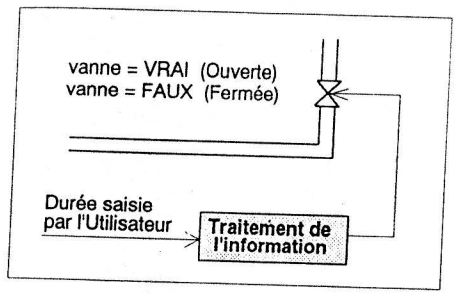


Fig. B-9. Saisie pour Boucle d'attente

a) Algorithme

```
Début algorithme attente
   entier : duree, min = 5000, max = 30000
   logique : vanne
Début action
   écrire 'Saisir la durée d''ouverture de vanne '
   lire duree
   Tant que duree < min OU duree > max faire
      écrire 'Saisir durée de ',min, ' à ',max
      lire duree
   Fin Tant que
   vanne <--- VRAI      /* Ouverture de Vanne */
   écrire 'Vanne ouverte'
   Tant que duree > 0 faire  /* Boucle d'attente */
      duree <--- duree - 1
   Fin Tant que
   vanne <--- FAUX      /* Fermeture de Vanne */
   écrire 'Vanne fermée'
Fin action
Fin algorithme
```

Remarque 1 : La première boucle Tant que effectue un contrôle de validité sur la saisie de l'utilisateur, entre les bornes *min* et *max*.

Remarque 2 : Le programme ci-contre montre que dans la première instruction WHILE, chaque condition, de part et d'autre de l'opérateur doit être entre parenthèses. La seconde instruction WHILE s'écrit sur une ligne.

b) Programme Pascal

```
PROGRAM attente (INPUT, OUTPUT);
CONST   min = 5000; max = 30000;
VAR     duree : INTEGER;
        vanne : BOOLEAN;
BEGIN
   WRITE('Saisir la duree d''ouverture de vanne ');
   READLN(duree);
   WHILE (duree < min) OR (duree > max) DO
      BEGIN
      WRITELN('Saisir temps de ',min, ' à ',max);
      READLN(duree)
      END;
   vanne := TRUE;          (* Ouverture de Vanne *)
   WRITELN('Vanne ouverte');
   WHILE duree > 0 DO duree := duree - 1;
   vanne := FALSE;         (* Fermeture de Vanne *)
   WRITELN('Vanne fermee')
END.
```

APPLICATION III-4

En fin de fabrication, les trois types de produits fabriqués sont conditionnés dans des caisses de hauteur différente : basse, moyenne et haute (Fig. B-10). Ces caisses sont ensuite convoyées par un transporteur à bande.
Un poste de comptage équipé de trois cellules photoélectriques, **C_b, C_m et C_h**, disposées verticalement, détecte la nature des produits convoyés, en fonction de la hauteur des caisses.
A partir de cette saisie d'information, une unité de traitement calcule, par rapport au nombre total de produits détectés, le pourcentage correspondant à chaque type de produit fabriqué. Le comptage commence dès la mise en marche du convoyeur et prend fin avec l'arrêt du convoyeur.
Etablir l'algorithme et le programme de la partie commande de ce système automatisé.

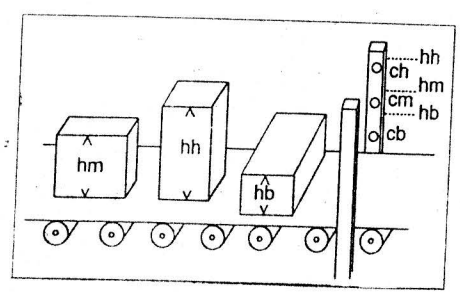


Fig. B-10. Détection et comptage

a) Algorithme

```
Début algorithme comptage
  entier :   nbbas, nbmed, nbhau,nbtot,
  /* nombre produits bas,médians,hauts,total */
             basse, mediane, haute
  /* valeur 1 si cellule activée, sinon valeur 0 */
  caractère : controle /* Marche Arrêt convoyeur*/
Début action
  nbbas <--- 0, nbmed <--- 0
  nbhau <--- 0, nbtot <--- 0   /* initialisations */
  écrire 'Controle Marche du convoyeur (M/A)'
  lire controle
  Tant que controle = 'M' faire
    écrire 'Lire cellule basse (0/1) 0 = pièce'
    lire basse
    /* Les cellules médiane et haute sont lues */
    /* seulement si basse n'est plus alimentée */
    si basse = 0      /* cellule non alimentée */
      alors
        nbtot <--- nbtot + 1
        écrire 'Lire cellule médiane'
        lire mediane
        si mediane = 0
          alors
            écrire 'Lire cellule haute'
            lire haute
            si haute = 0
              alors nbhau <--- nbhau + 1
              sinon nbmed <--- nbmed +1
            Fin si
          sinon nbbas <--- nbbas + 1
        Fin si
    /* A cet instant l'objet est devant les cellules*/
    /* et il ne faut plus faire de comptage avant */
    /* que cellule basse soit alimentée (= 1) */
    /* donc une boucle d'attente s'impose */
        Tant que basse =0 et controle ='M' faire
          écrire 'Attente cell. basse a 1'
          lire basse
          écrire 'Convoyeur (M/A)'
          lire controle
        Fin Tant que
    Fin si
    si controle = 'M' alors
      écrire 'Controle convoyeur (M/A)'
      lire controle
    Fin si
  Fin Tant que
  si nbtot <> 0 alors
    écrire 'POURCENTAGE PRODUITS'
    écrire 'produits bas = ',100 * nbbas / nbtot
    écrire 'prod. médians = ',100 * nbmed / nbtot
    écrire 'produits hauts = ',100 * nbhau / nbtot
  Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM comptage (INPUT,OUTPUT);
VAR
  nbbas, nbmed, nbhau, nbtot,
  basse, mediane, haute : INTEGER;
  controle : CHAR;
BEGIN
  nbbas := 0; nbmed := 0; nbhau := 0; nbtot := 0;
  WRITE('Controle Marche du convoyeur (M/A) ');
  READLN(controle);
  WHILE controle = 'M' DO
  BEGIN
    WRITE('Lire cellule basse (0/1) 0 = piece) ');
    READLN(basse);
    IF basse = 0 THEN (*Cellule non alimentee*)
      BEGIN
        nbtot := nbtot + 1;
        WRITE('Lire cellule mediane ');
        READLN(mediane);
        IF mediane = 0 THEN
          BEGIN
            WRITE('Lire cellule haute ');
            READLN(haute);
            IF haute = 0
              THEN nbhau := nbhau + 1
              ELSE nbmed := nbmed +1
          END
          ELSE nbbas := nbbas + 1;
          WHILE (basse = 0) AND
                (controle = 'M') DO
            BEGIN
              WRITE('Attente cell. basse a 1');
              READLN(basse);
              WRITE('Convoyeur (M/A) ');
              READLN(controle)
            END
      END;
    IF controle = 'M' THEN
      BEGIN
        WRITELN('Controle convoyeur (M/A)');
        READLN(controle)
      END
  END;
  IF nbtot <> 0 THEN
    BEGIN
    WRITELN('POURCENTAGES PRODUITS');
    WRITELN('bas = ', 100*nbbas/nbtot :3:2);
    WRITELN('medians=',100*nbmed/nbtot:3:2);
    WRITELN('hauts = ', 100*nbhau/nbtot :3:2)
    END
END.
```

EXERCICE III-3 : $e^x = 1 + x + x^2/2! + ... + x^n/n!$
Après saisie de x, Calculer e^x en arrêtant le calcul dès que le terme $x^i/i!$ devient supérieur à 0.0001.

10. STRUCTURE REPETER...

Considérons une file dont le nombre d'éléments est inconnu et supposons que l'élément de queue doive subir le même traitement que ses prédécesseurs. Dans ce cas, **on répète le traitement jusqu'à l'élément de queue, c'est à dire jusqu'à ce que la condition correspondant à l'élément de queue soit égale à VRAI.**

Notation algorithmique et Organigramme

```
Répéter
  action
Jusqu'à <condition>
```

L'action précède toujours le test. Elle est d'abord exécutée, puis la condition est testée. Si elle n'est pas satisfaite, l'action est reconduite à nouveau.

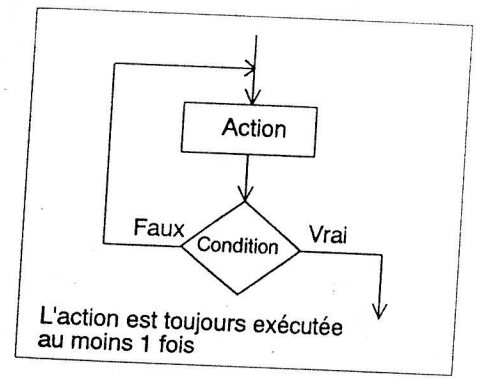
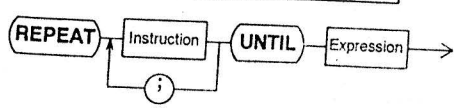


Fig. B-11. Structure REPETER

Syntaxe Pascal et Diagramme syntaxique

```
REPEAT <instruction> UNTIL
<expression logique>
```



L'instruction est toujours exécutée au moins une fois puis l'expression de type booléen est évaluée. Si elle fournit un résultat FALSE (FAUX), l'instruction est exécutée à nouveau. Lorsque le résultat est TRUE (VRAI), on sort de la boucle sans exécuter l'instruction.

L'instruction contenue dans la structure REPEAT, doit pouvoir modifier l'expression, qui doit fournir un résultat TRUE (VRAI), après un certain nombre d'itérations. **Dans le cas contraire, la boucle serait infinie.**

■ EXEMPLE NUMERIQUE SIMPLE

Algorithme et Programme permettant la saisie par l'utilisateur d'une série de nombres entiers. **La saisie est terminée lorsque la somme des nombres devient supérieure ou égale à 1000. Le dernier nombre, élément de queue de la file, sera donc traité.**

a) Algorithme

```
Début algorithme somme3
  entier : nombre, som
Début action
  écrire 'Saisie jusqu"à somme >= 1000'
  som <--- 0
  Répéter
    écrire 'Saisir un nombre entier '
    lire nombre
    som <--- som + nombre
  Jusqu'à som >= 1000
  écrire 'somme = ', som : 5
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM somme3 (INPUT,OUTPUT);
VAR   nombre, som : INTEGER;
BEGIN
  som := 0;
  WRITELN('Saisie jusqu"a somme >= 1000');
  REPEAT
    WRITE('Saisir un nombre entier ');
    READLN(nombre);
    som := som + nombre
  UNTIL som >= 1000;
  WRITELN('somme = ', som : 5)
END.
```

Remarque 1 : On ne place jamais de point virgule devant le mot UNTIL.
Remarque 2 : Le corps du REPEAT contient une instruction composée non encadrée par un couple BEGIN END.

APPLICATION III-5

Contrôle de Saisie : Demander à l'utilisateur de saisir une borne inférieure, une borne supérieure et une valeur comprise entre ces 2 bornes. Les contrôles de saisie porteront sur la borne supérieure (qui devra être supérieure à la borne inférieure), ainsi que sur la valeur bornée. Puis afficher la valeur, encadrée par les bornes.

a) Algorithme

```
Début algorithme saisie
   entier : borneinf, bornesup, val
Début action
   écrire 'Saisir borne inférieure'
   lire borneinf
   Répéter
      écrire 'Saisir borne supérieure'
      lire bornesup
   Jusqu'à bornesup > borneinf
   Répéter
      écrire 'Saisir la valeur entre les 2 bornes'
      lire val
   Jusqu'à val >= borneinf et val <= bornesup
   écrire borneinf, ' <= ', val, ' <= ', bornesup
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM saisie (INPUT, OUTPUT);
VAR
   borneinf, bornesup, val : INTEGER;
BEGIN
   WRITELN('Saisir borne inferieure');
   READLN(borneinf);
   REPEAT
      WRITELN('Saisir borne superieure');
      READLN(bornesup)
   UNTIL bornesup > borneinf;
   REPEAT
      WRITE('Saisir la valeur entre les 2 bornes ');
      READLN(val);
   UNTIL (val >= borneinf) AND (val <=bornesup);
   WRITELN(borneinf, ' <= ', val, ' <= ', bornesup)
END.
```

Rappel 1 : On ne place jamais de point virgule devant le mot UNTIL.
Rappel 2 : Le corps des REPEAT contiennent une instruction composée non encadrée par un couple <BEGIN END>.

APPLICATION III-6

Remplissage contrôlé d'une cuve avec régulation par un flotteur et des contacteurs de niveau. La cuve étant vide, le débit commandé par la vanne est maximum et égal à 100. En fonction du niveau atteint (1, 2, 3, 4), le débit diminue à 90, 70, 40 puis 10. Au niveau 5, la vanne est fermée, le débit est égal à 0, et le remplissage est terminé.

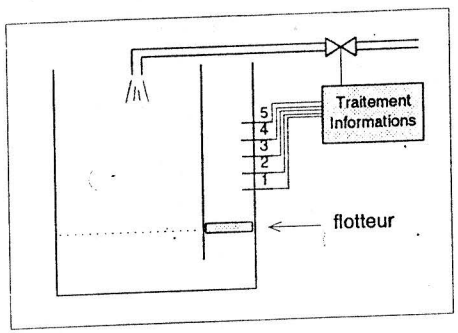


Fig. B-12. Régulation de débit

a) Algorithme

```
Début algorithme régulation
   entier : niveau, debit
Début action
   écrire 'cuve vide, débit maximum'
   debit <--- 100
   Répéter
      écrire 'lecture du niveau'
      lire niveau
      selon que niveau
         1 : debit <--- 90
         2 : debit <--- 70
         3 : debit <--- 40
         4 : debit <--- 10
         5 : debit <--- 0
      Fin selon
      écrire 'vanne ouverte a ', debit
   Jusqu'a debit = 0
   écrire 'remplissage terminé'
Fin action
Fin algorithme
```


b) Programme Pascal

```
PROGRAM regulation (INPUT,OUTPUT);
VAR   niveau, debit : INTEGER;
BEGIN
   WRITELN('cuve vide, debit maximum');
   debit := 100;
   REPEAT
      WRITELN('lecture du niveau');
      READLN(niveau);
      CASE niveau OF
         1 : debit := 90;
         2 : debit := 70;
         3 : debit := 40;
         4 : debit := 10;
         5 : debit := 0
      END;
      WRITELN('vanne ouverte a ', debit)
   UNTIL debit = 0;
   WRITELN('remplissage termine')
END.
```

APPLICATION III-7

Mise à température d'un liquide contenu dans une cuve. Le chauffage n'est déclenché que si le liquide est au dessous d'une température minimum, et s'arrête lorsqu'il atteint une température maximum.

a) Algorithme

```
Début algorithme chauffage
   entier : chauffe      /* 0 = Arrêt, 1 = Marche */
   réel : tmplue, tmpmini = 47.5, tmpmaxi = 52.5
   /* température lue, mini = 47.5, maxi = 52.5 */
Début action
   écrire 'Mesure température'
   écrire 'chauffage si <  ',tmpmini :4:2
   lire tmplue
   si tmplue < tmpmini
      alors
         écrire 'Mise en marche du chauffage'
         chauffe <--- 1
         Répéter
            écrire 'Mesure température'
            écrire 'arrêt si >  ',tmpmaxi :4:2
            lire tmplue
         Jusqu'à tmplue >= tmpmaxi
         écrire 'Arrêt du chauffage'
         chauffe <--- 0
   Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM chauffage (INPUT, OUTPUT);
CONST
   tmpmini = 47.5; tmpmaxi = 52.5;
VAR
   chauffe : INTEGER;
   tmplue : REAL;
BEGIN
   WRITELN('Mesure temperature');
   WRITELN('chauffage si <  ',tmpmini :4:2);
   READLN(tmplue);
   IF tmplue < tmpmini
      THEN
      BEGIN
         WRITELN('Mise en marche chauffage');
         chauffe := 1;
         REPEAT
            WRITELN('Mesure temperature');
            WRITELN('arret si >  ',tmpmaxi :4:2);
            READLN(tmplue)
         UNTIL tmplue >= tmpmaxi;
         WRITELN('Arret du chauffage');
         chauffe := 0
      END
END.
```

APPLICATION III-8

Ajustement du remplissage à 50 kg d'un sac, avec une précision donnée par l'utilisateur entre 0.1 kg et 0.5 kg. Le sac est placé sur un dispositif de pesée. Le fonctionnement des vannes 1 et 2 est décrit après la Figure B-13.

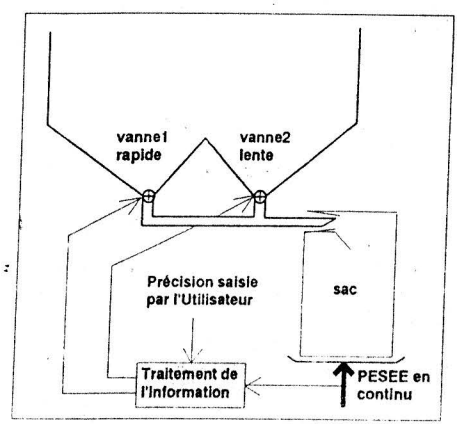


Fig. B-13. Pesage automatisé

Une première électrovanne rapide déverse 47 kg (en 1000 unités de temps) plus ou moins 3 kg dans le sac. Une seconde électrovanne lente, pilotée par la pesée en continu, injecte le complément dans le sac.

a) Algorithme

```
Début algorithme ensachage
  entier : temporis,
           /* pour temporisation */
           install
           /* 1 = action placement de sac vide */
           /* 0 = action éjection du sac plein */
  logique : vanne1, vanne2
           /* VRAI = ouverture vanne */
           /* FAUX = fermeture vanne */
  réel : poids, precis   /* précision */
Début action
  Répéter
    écrire 'Saisir la précision de 0.1 a 0.5'
    lire precis
  Jusqu'à precis >= 0.1 et precis <= 0.5
  écrire 'installation d''un sac vide'
  install <--- 1
  temporis <--- 1000
  écrire 'ouverture 1ère électrovanne'
  vanne1 <--- VRAI
  Répéter
    temporis <--- temporis - 1
  Jusqu'à temporis = 0
  écrire 'fermeture 1ère électrovanne'
  vanne1 <--- FAUX
  écrire 'mesure du poids de 44 a 50'
  lire poids
  si poids < 50 - precis
  alors
    écrire 'ouverture 2ème électrovanne'
    vanne2 = VRAI
    Répéter
      écrire 'mesure du poids'
      lire poids
    Jusqu'à poids > 50 - precis et
          poids < 50 + precis
    écrire 'fermeture 2ème électrovanne'
    vanne2 <--- FAUX
  Fin si
  écrire 'éjection du sac'
  install <--- 0
Fin action
Fin algorithme
```

Remarque : Les variables *install*, *vanne1*, et *vanne2*, prennent ici des valeurs qui correspondent à des ordres. Ces ordres provoquent des actions physiques commandées par d'autres processeurs :

- Placement ou Ejection d'un sac,
- Ouverture ou Fermeture d'une vanne.

b) Programme Pascal

```
PROGRAM ensachage (INPUT,OUTPUT);
VAR
  temporis, install : INTEGER;
  vanne1, vanne2 : BOOLEAN;
  poids, precis : REAL;
BEGIN
  REPEAT
    WRITELN('Saisir la precision de 0.1 a 0.5');
    READLN(precis)
  UNTIL (precis >= 0.1) AND (precis <= 0.5);
  WRITELN('installation d''un sac vide');
  install := 1;
  temporis := 1000;
  WRITELN('ouverture 1ere electrovanne');
  vanne1 := TRUE;
  REPEAT
    temporis := temporis - 1
  UNTIL temporis = 0;
  WRITELN('arret 1ere electrovanne');
  vanne1 := FALSE;
  WRITELN('mesure du poids de 44 a 50');
  READLN(poids);
  IF poids < 50 - precis THEN
    BEGIN
      WRITELN('ouverture 2eme electrovanne');
      vanne2 := TRUE;
      REPEAT
        WRITELN('mesure du poids');
        READLN(poids)
      UNTIL (poids > 50 - precis) AND
            (poids < 50 + precis);
      WRITELN('fermeture 2eme electrovanne');
      vanne2 := FALSE
    END;
  WRITELN('ejection du sac');
  install := 0
END.
```

EXERCICE III-4 : Une cellule circulaire de diamètre D (compris entre 0.1μ et 0.7μ) se reproduit en triplant de surface toutes les 10 heures. Au bout de combien de jours les cellules couvriront-elles une surface de 1 cm^2 ? La condition d'arrêt est Surface >= 1 cm^2.

IV - PROCEDURES ET FONCTIONS

11. ANALYSE DESCENDANTE

Un processeur est une entité capable de comprendre et d'exécuter un nombre défini d'actions élémentaires appartenant à son jeu d'instructions. Ces actions élémentaires sont également appelées primitives.
Une action primitive pour un processeur donné est une action dont l'énoncé est suffisant en lui-même pour que le processeur puisse exécuter l'action sans autre information.

Comment peut-on demander à un processeur de réaliser une action complexe pour laquelle il n'a pas de réponse immédiate ?

L'analyse descendante permet de traduire une action complexe par décompositions successives en actions de plus en plus simples, ne contenant finalement que des actions primitives.

Soit une action A à réaliser, décrite par un énoncé non primitif. L'analyse descendante consiste à trouver une décomposition A1, A2, A3 ... An, qui soit une suite d'actions réalisant l'action A.
La figure B-14 fait apparaître les niveaux hiérarchiques de décomposition de l'action A. Les actions à décomposer sont inscrites dans des cercles, tandis que les actions primitives sont inscrites dans des rectangles grisés.

Toute action Ai est :

- **Soit une action primitive qui est terminale de l'analyse descendante sur cette branche,**
- **Soit une action non primitive qui doit être décomposée en Ai1, Ai2, ..., Aix, pour aboutir à un ensemble d'actions primitives.**

Cette méthode est également utilisée pour analyser un système, dont la conception met en évidence plusieurs niveaux, telle la Méthode S.A.D.T. (Structured Analysis and Design Technic), marque déposée par SOFETECH et IGL Technology France.

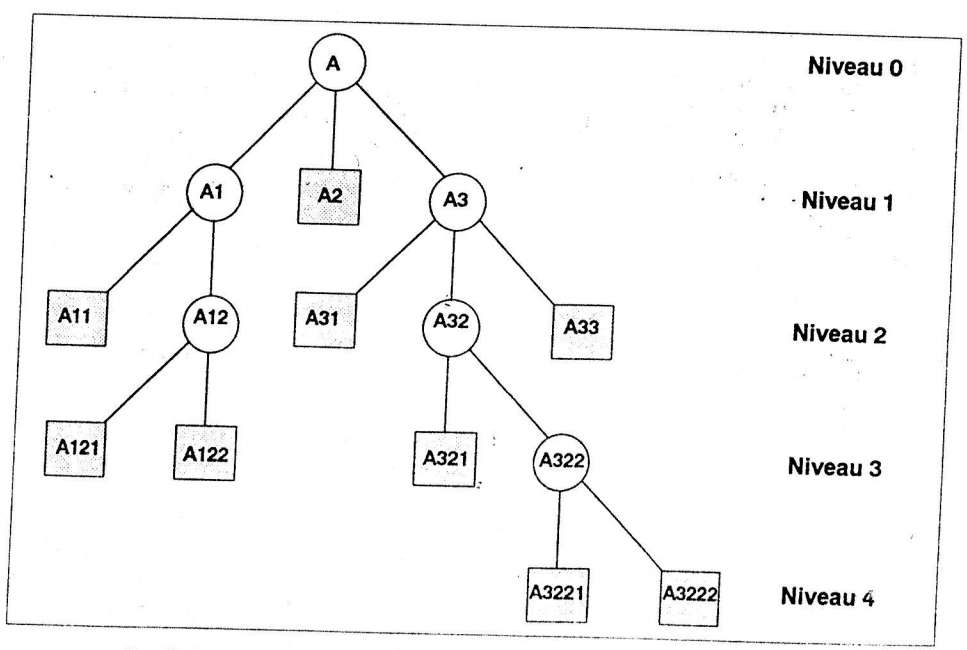


Fig. B-14. Analyse descendante de l'action complexe A sur 4 niveaux

L'exécution de l'ensemble des actions primitives de la figure B-15 réalise l'action A.

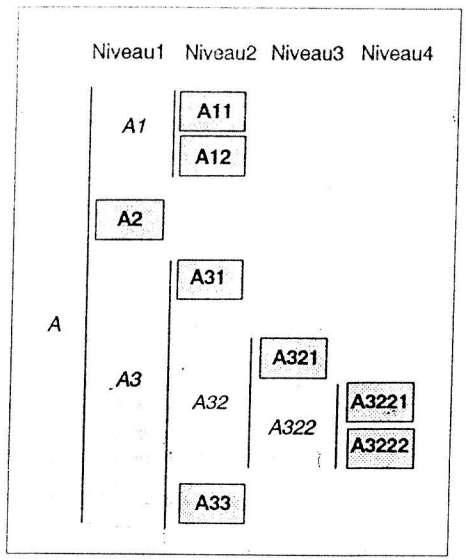


Fig. B-15. Réalisation de l'action A

L'algorithmique utilise l'analyse descendante dans l'écriture d'actions complexes, décrites dans des sous-programmes, appelés actions nommées.

A la nécessité pour le processeur, de voir décomposer une action complexe en actions primitives, se joint **l'intérêt de cette méthode, pour l'analyste du problème, lui permettant de structurer la solution, en partant des éléments les plus généraux pour aller vers les éléments de détails particuliers.**

L'analyse descendante d'un problème permet de structurer la solution en hiérarchisant les niveaux de décompositions successives. L'algorithme réalise une synthèse qui ordonne les différentes actions de manière séquentielle.
Comme le montre la figure B-16, chaque action est clairement libellée par un début d'action et une Fin d'action qui délimitent un bloc. Les blocs inclus les uns dans les autres représentent les actions nommées de l'algorithme.

Le bloc A contient les blocs A1, A2, et A3 qui sont, dans l'ordre, les actions à exécuter pour réaliser l'action A. De même, les blocs A11 et A12, imbriqués dans le bloc A1, réalise l'action A1.

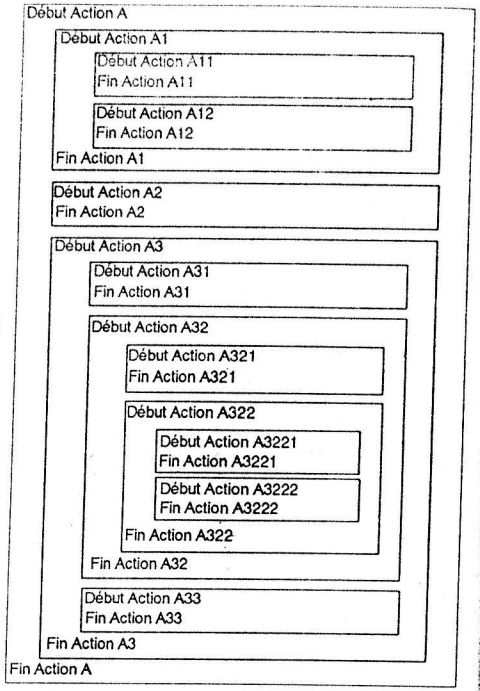


Fig. B-16. Structure algorithmique de l'action A

L'écriture d'un algorithme fait apparaître des actions nommées qui peuvent être appelées une ou plusieurs fois. Dans l'algorithme suivant, l'action 1, dont le contenu ne sera détaillé qu'une seule fois dans l'action nommée, est appelée deux fois par le corps de l'algorithme (Fig. B-17).

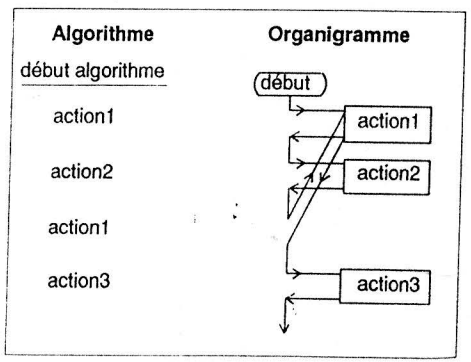


Fig. B-17. Appel répété d'une action nommée.

12. PROCEDURES

Le Pascal reprend ces structures d'actions nommées et les enrichit au travers des procédures et des fonctions. Leur utilisation permet au programmeur d'améliorer la lisibilité du programme, d'en faciliter la maintenance et de n'avoir à écrire qu'une seule fois dans une procédure ou une fonction, une action appelée plusieurs fois par le programme.

1°- DEFINITION ET DECLARATION

Dans l'algorithme, l'action nommée se présente sous la forme :

```
Action nomaction(paramètres)
    partie des déclarations
    Début action
        action1
        actionn
    Fin action
```

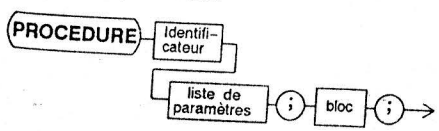
Une procédure est un sous-programme réalisant une action sur une partie des données du programme. La structure d'une procédure est identique à celle d'un programme. Elle est constituée d'un en-tête et d'un bloc.

Syntaxe Pascal et Diagramme syntaxique

```
<procédure> ::= <en-tête> <bloc>

<en-tête de procédure> ::=
        PROCEDURE <identificateur>
                  <liste de paramètres>
```

L'en-tête commence par le mot **PROCEDURE** puis identifie la procédure par son identificateur, nom donné par le programmeur, et fournit enfin une liste de paramètres.



L'identificateur de la procédure est utilisé, dans le programme appelant la procédure, comme instruction d'appel ou d'activation de la procédure.

Les paramètres sont utilisés dans le cas où la communication des informations entre le programme appelant et la procédure doit se réaliser sous une forme spécifique. Nous reviendrons plus tard sur les modes de passage des paramètres.
La liste de paramètres peut être vide.

Le bloc de la procédure est structuré comme la partie bloc d'un programme.

On peut donc retrouver les parties :
* déclaration d'étiquettes,
* définition de constantes,
* définition de types,
* définition de variables,
* procédures et fonctions
* instructions exécutables.

La seule partie obligatoire du bloc est celle des instructions exécutables.

Dans la mesure où la partie procédures et fonctions n'est pas vide, une procédure pourra être incluse dans une autre procédure. Nous verrons plus loin l'intérêt de ces procédures imbriquées.

2°- PROCEDURES SANS PARAMETRE

Les Règles de mise en oeuvre des actions nommées de l'algorithme et des procédures du Pascal diffèrent sur un point : celui de l'emplacement par rapport à l'instruction d'appel.

Dans l'algorithme, l'analyste du problème écrit d'abord les actions principales qui sont souvent des actions complexes, puis va écrire le détail de ces actions dans les actions nommées. Cette façon de procéder respecte parfaitement le principe de l'analyse descendante selon lequel la traduction d'une action complexe se fait par décompositions successives. La déclaration et l'écriture d'une action nommée figure donc après son appel.

Une règle du langage Pascal prévoit au contraire qu'une procédure ne peut être appelée que si elle a été déclarée, c'est à dire connue, avant l'instruction d'appel.

Cette différence est mise en évidence dans l'exemple numérique suivant.

■ EXEMPLE NUMERIQUE SIMPLE

Calculer l'expression A+B-C, par une *procédure calculABC*, pour 2 valeurs différentes de A, B, C.

a) Algorithme

```
Début algorithme exempleproc
   entier : A, B, C, resultat
Début action principale
   A <--- 21
   B <--- 9
   C <--- 5
   écrire 'Premier appel'
   calculABC
   A <--- 6, B <--- 14, C <--- 8
   écrire 'Second appel'
   calculABC
Fin action principale

Action calculABC
Début action
   resultat <--- A+B-C
   écrire 'Valeur = ',resultat : 3
Fin action

Fin algorithme
```

b) Programme Pascal

```
PROGRAM exempleproc (OUTPUT);
VAR   A,B,C,resultat : INTEGER;

PROCEDURE calculABC;
BEGIN
   resultat := A+B-C;
   WRITELN('Valeur = ',resultat);
END;

BEGIN                    (* Programme principal*)
   A := 21;
   B := 9;
   C := 5;
   WRITE('Premier appel  ');
   calculABC;
   A := 6; B := 14; C := 8;
   WRITE('Second appel   ');
   calculABC
END.
```

c) Affichage écran

```
Premier appel  Valeur = 25
Second appel   Valeur = 12
```

Remarque : L'action nommée *calculABC* apparaît après l'action principale de l'algorithmique. La procédure *calculABC* apparaît avant le programme principal du programme. Le sous-programme *calculABC* est appelé 2 fois avec des valeurs différentes, pour effectuer le même calcul.

APPLICATION IV-1

Modifier l'algorithme et le programme de l'application III-7 (Mise à température d'un liquide contenue dans une cuve), en intégrant une **procédure de lecture de température.**

a) Algorithme

```
Début algorithme chauffage2
   entier : chauffe
         /* 0 = Arrêt, 1 = Marche */
   réel : tmplue, tmpmini = 47.5, tmpmaxi = 52.5
         /* températures lue et */
         /* minimum = 47.5 et maximum = 52.5 */

Début action principale
   mesurer
   si tmplue < tmpmini
      alors
         écrire 'Mise en marche du chauffage'
         chauffe <--- 1
         Répéter
            mesurer
         Jusqu'à tmplue >= tmpmaxi
         écrire 'Arrêt du chauffage'
         chauffe <--- 0
   Fin si
Fin action principale

Action mesurer
Début action
   écrire 'chauffage si <  ',tmpmini :4:2
   écrire 'arrêt chauffage si >  ',tmpmaxi :4:2
   écrire 'Mesure température'
   lire tmplue
Fin action

Fin algorithme
```


b) Programme Pascal

```
PROGRAM chauffage2 (INPUT, OUTPUT);
CONST
      tmpmini = 47.5; tmpmaxi = 52.5;
VAR
      chauffe : INTEGER;
      tmplue : REAL;

PROCEDURE mesurer;
BEGIN
   WRITELN('chauffage si <  ',tmpmini :4:2);
   WRITELN('arret chauffage si >  ',tmpmaxi :4:2);
   WRITELN('Mesure temperature');
   READLN(tmplue)
END;

BEGIN                    (* programme principal *)
   mesurer;
   IF tmplue < tmpmini
   THEN
      BEGIN
      WRITELN('Mise en marche du chauffage');
      chauffe := 1;
      REPEAT mesurer
      UNTIL tmplue >= tmpmaxi;
      WRITELN('Arret du chauffage');
      chauffe := 0
      END
END.
```

Remarque 1 : L'action nommée *mesurer* apparaît après l'action principale. La procédure *mesurer* apparaît avant le programme principal.

Remarque 2 : Le sous-programme *mesurer* est appelé une première fois au début de l'action principale, puis autant de fois que nécessaire dans la boucle Répéter. La procédure *mesurer* ne contient qu'une partie instructions exécutables. L'en-tête de procédure est sous la forme la plus simple possible, en n'utilisant pas de paramètres.

Remarque 3 : On notera la syntaxe simplifiée du REPEAT qui ne fait appel qu'à une instruction unique : l'appel de la procédure *mesurer.*

IMPORTANT

Lorsqu'un sous-programme (procédure ou fonction) intègrera une boucle FOR, l'indice du compteur de boucle devra être déclaré comme variable dans ce sous-programme.

APPLICATION IV-2

Reprendre l'application II-2 ordonnant 2 variables et modifier l'algorithme et le programme en mettant en oeuvre **une procédure de saisie des données et une procédure d'échange.**

a) Algorithme

```
Début algorithme ordonner2
   réel :petit, grand,tampon
   logique : ordrecorrect

Début action principale
   lirevaleur
   si NON ordrecorrect
      alors échange
   Fin si
   écrire 'Valeurs ordonnées :  '
   écrire petit :4:2, '  ', grand :4:2
Fin action principale

Action lirevaleur
Début action
   écrire 'Saisir la 1ère valeur  '
   lire petit
   écrire 'Saisir la 2ème valeur  '
   lire grand
   ordrecorrect <--- petit < grand
Fin action

Action échange
Début action
   tampon <--- petit
   petit <--- grand
   grand <--- tampon
Fin action

Fin algorithme
```

Remarque : L'algorithme *ordonner* contient maintenant trois actions : l'action principale et les actions nommées *lirevaleur* et *échange.* Les actions nommées sont souvent identifiées par des verbes d'action.

b) Programme Pascal

```
PROGRAM ordonner2 (INPUT,OUTPUT);
VAR
    petit, grand, tampon : REAL;
    ordrecorrect : BOOLEAN;

PROCEDURE lirevaleur;
BEGIN
  WRITE('Saisir la 1ere valeur ');
  READLN(petit);
  WRITE('Saisir la 2eme valeur ');
  READLN(grand);
  ordrecorrect := petit < grand
END;

PROCEDURE echange;
BEGIN
  tampon := petit;
  petit := grand;
  grand := tampon
END;

BEGIN                    (* programme principal *)
  lirevaleur;
  IF NOT ordrecorrect THEN echange;
  WRITELN('Valeurs ordonnees : ');
  WRITELN(petit :4:2, '  ', grand :4:2)
END.
```

Remarque 1 : L'analyste du problème ne doit pas hésiter à écrire des procédures, même si elles ne sont utilisées qu'une seule fois. L'utilisation des procédures améliore la lisibilité et la recherche d'éventuelles erreurs de programmation.

Remarque 2 : Le programme principal est simplifié. Son exécution suit les points suivants :

* L'instruction *lirevaleur* appelle la procédure *lirevaleur*, lui passe le contrôle du déroulement, et *lirevaleur* est exécutée.
* Le programme reprend son enchaînement à l'instruction suivante, qui est le test de la variable logique *ordrecorrect*. Si NOT *ordrecorrect* est VRAI (*ordrecorrect* est FAUX), l'instruction *echange* appelle la procédure *echange*.
* Le programme reprend et termine son enchaînement par l'instruction suivante, qui est l'écriture des valeurs ordonnées.

Remarque 3 : Les procédures ne contiennent ici que leur partie instructions exécutables. Les en-têtes de procédures sont sous la forme la plus simple possible, en n'utilisant pas de paramètres.

Remarque 4 : Les procédures *échange* et *lirevaleur* sont de même niveau par rapport au programme principal qui les appelle : on pourrait intervertir leur ordre d'apparition dans le programme.

■ PORTEE DES VARIABLES

Dans les 2 applications précédentes, toutes les variables et les constantes sont déclarées en tête d'algorithme et de programme.
Ces données sont reconnues et manipulées aussi bien par l'action principale que par les actions nommées ou procédures. Elles ont une portée globale sur l'algorithme et le programme. On parle dans ce cas de ***variables globales***.

Les 2 applications suivantes vont faire apparaître des variables déclarées dans la partie déclaration de variables, au sein des procédures. Ces variables ne seront reconnues et manipulées que par la procédure qui les met en jeu. On parle dans ce cas de ***variables locales***.

APPLICATION IV-3

Reprendre l'application IV-2 en appelant l'action *échange* à partir de l'action *lirevaleur*. **Etude de la portée des variables et mise en oeuvre de variables locales** nécessaires aux actions nommées de l'algorithme et aux procédures du programme.

a) Analyse de portée des variables.

– Les variables *petit* et *grand*, utilisées par toutes les actions, doivent être déclarées comme variables globales.
– La variable *ordrecorrect* n'est manipulée que dans l'action *lirevaleur*. Elle est donc déclarée localement dans *lirevaleur*.
– La variable *tampon* n'intervient que dans l'action *échange*, il est donc inutile de la déclarer ailleurs que dans l'action *échange*.

b) Algorithme

```
Début algorithme ordonner3
  réel :petit, grand
Début action principale
  lirevaleur
  écrire 'Valeurs ordonnées : '
  écrire petit :4:2, '  ', grand :4:2
Fin action principale

Action lirevaleur
  logique : ordrecorrect
Début action
  écrire 'Saisir la 1ère valeur '
  lire petit
  écrire 'Saisir la 2ème valeur '
  lire grand
  ordrecorrect <--- petit < grand
  si NON ordrecorrect
    alors échange
  Fin si
Fin action

Action échange
  réel : tampon
Début action
  tampon <--- petit
  petit <--- grand
  grand <--- tampon
Fin action
Fin algorithme
```

Remarque : L'algorithme *ordonner* contient toujours trois actions, mais l'action *échange* est appelée non plus à partir de l'action principale mais à partir de l'action *lirevaleur*. Les places respectives des actions *échange* et *lirevaleur* ne changent pas dans l'algorithme qui, opérant par analyse descendante, décompose le niveau le plus élevé vers le niveau le plus bas.

c) Programme Pascal

```
PROGRAM ordonner3 (INPUT,OUTPUT);
VAR
  petit, grand : REAL;

PROCEDURE echange;
VAR  tampon : REAL;
BEGIN
  tampon := petit;
  petit := grand;
  grand := tampon
END;

PROCEDURE lirevaleur;
VAR
  ordrecorrect : BOOLEAN;
BEGIN
  WRITE('Saisir la 1ere valeur ');
  READLN(petit);
  WRITE('Saisir la 2eme valeur ');
  READLN(grand);
  ordrecorrect := petit < grand;
  IF NOT ordrecorrect THEN echange
END;

BEGIN (* programme principal *)
  lirevaleur;
  WRITELN('Valeurs ordonnees : ');
  WRITELN(petit :4:2, '  ', grand :4:2)
END.
```

Remarque 1 : Les places respectives des procédures *échange* et *lirevaleur* ont été modifiées : La procédure *echange* précède la procédure *lirevaleur*, car en Pascal une procédure doit être définie avant d'être appelée.

Remarque 2 : La variable *tampon* est locale à la procédure *echange*. La variable *ordrecorrect* est locale à la procédure *lirevaleur*. Les variables *petit* et *grand* sont globales.

Remarque 3 : Le programmeur aura toujours intérêt à déclarer locaux à une procédure, tous les types, constantes, variables, et même procédure ou fonction, qu'il n'aura pas à utiliser en dehors de cette procédure.

d) Intérêt de la Portée des variables

Les variables mises en oeuvre dans une procédure peuvent se répartir en 3 sous ensembles disjoints :

* Variables vues seulement par la **procédure appelante,**
* Variables vues à la fois par la **procédure appelante et par la procédure appelée,**
* Variables vues seulement par la **procédure appelée.**

Le tableau suivant indique la portée des variables sur les différentes actions. Il mentionne 'oui' si la variable est visible par l'action et 'non' dans le cas contraire.

Action	*principale*	*lirevaleur*	*échange*
Variables			
petit	oui	oui	oui
grand	oui	oui	oui
ordrecorrect	non	oui	non
tampon	non	non	oui

La procédure *echange* n'a pas de visibilité sur la variable *ordrecorrect*, et ceci est voulu. Si on souhaitait la lui donner, sans définir globale la variable *ordrecorrect*, il faudrait inclure la procédure *echange* au sein de la procédure *lirevaleur* (dans la partie procédures et fonction) et réaliser ainsi une imbrication.

■ IMBRICATION DE PROCEDURES

Dans le programme *ordonner4*, ci-contre, la procédure *echange* est incluse, ou imbriquée dans la procédure *lirevaleur* et la variable *ordrecorrect* est visible par la procédure *echange*.

La mise en oeuvre des procédures imbriquées nécessite plus que jamais une attention particulière pour les commentaires qui doivent documenter le lecteur du programme et notamment lui permettre un bon repérage des début et Fin des procédures.

■ REGLE D'HOMONYMIE SUR LES VARIABLES

Lorsque dans une procédure, une variable locale est définie par un identificateur déjà utilisé, pour définir par exemple une variable globale (ou toute autre variable visible de cette procédure), la variable locale masque la variable globale qui devient invisible dans la procédure. La procédure ne pourra donc travailler que sur la variable homonyme, qui est locale.

EXERCICE IV-1 : Une personne effectuant un retrait à un distributeur de billet saisit au clavier le montant du retrait. Etablir une procédure qui, en fonction de l'état du compte, affiche le message 'Solde insuffisant - Demande trop élevée', ou débite le compte du montant demandé et affiche le nouveau solde.

```
PROGRAM ordonner4 (INPUT,OUTPUT);
VAR
  petit, grand : REAL;        (* variables globales *)

PROCEDURE lirevaleur;
VAR
  ordrecorrect : BOOLEAN;     (* variable locale *)
        (* à lirevaleur mais visible de echange *)

  PROCEDURE echange;
  VAR tampon : REAL;          (* locale à echange *)
  BEGIN                       (* Debut echange *)
    tampon := petit;
    petit := grand;
    grand := tampon
  END;                        (* Fin echange *)

BEGIN                         (* Debut lirevaleur *)
  WRITE('Saisir la 1ere valeur  ');
  READLN(petit);
  WRITE('Saisir la 2eme valeur  ');
  READLN(grand);
  ordrecorrect := petit < grand;
  IF NOT ordrecorrect THEN echange
END;                          (* Fin lirevaleur *)

BEGIN                 (* Debut programme principal *)
  lirevaleur;
  WRITELN('Valeurs ordonnees :  ');
  WRITELN(petit :4:2, '  ', grand :4:2)
END.                  (* Fin programme principal *)
```

■ PORTEE DES IDENTIFICATEURS

Ce que nous venons de dire à propos de la portée des variables est vrai pour tout identificateur, qu'il soit de variable, de constante, de type mais aussi pour les identificateurs de procédures, et donc les procédures elles-mêmes.

Une procédure peut être visible ou invisible d'une autre procédure. Elle ne peut être appelée que si elle est visible. Ainsi, dans le programme *ordonner4*, la procédure echange est totalement invisible du programme principal.

EXERCICE IV-2 : Un nombre N est saisi au clavier par l'utilisateur. Ecrire une procédure calculant et affichant les diviseurs de ce nombre (entre 2 et N/2).

3°- PROCEDURES AVEC PARAMETRES

Les sous-programmes appelant et appelé peuvent avoir besoin de se communiquer des informations. Cette communication est établie à l'aide de paramètres qui vont apparaître à la fois dans l'instruction d'appel et dans l'en-tête du sous-programme appelé (action nommée de l'algorithme ou procédure du programme).
Nous avons déjà rencontré deux procédures standards acceptant des paramètres : Les procédures READ et WRITE sont des procédures prédéfinies du langage Pascal.

Lorsque l'en-tête du sous-programme contient des paramètres, ceux-ci portent le nom de ***paramètres formels***. Ils sont représentés par des identificateurs sur lesquels le sous-programme va travailler et qui sont locaux au sous-programme.

Lorsque l'instruction d'appel contient des paramètres, ceux-ci portent le nom de ***paramètres effectifs***. Ils communiquent des informations au sous-programme appelé, dans lequel ils viennent se substituer aux paramètres formels.

Les informations communiquées entre sous-programmes peuvent être des valeurs, des variables, des procédures ou des fonctions.
- Chaque appel doit comprendre autant de paramètres effectifs que l'en-tête du sous-programme contient de paramètres formels.
- Les paramètres effectifs doivent être de même type que les paramètres formels auxquels ils correspondent.

Dans un programme, les sous-programmes paramétrés peuvent mettre en oeuvre trois catégories de paramètres :

- Le paramètre DONNEE ou PAR VALEUR, dont la valeur reste inchangée dans le sous-programme appelant après exécution du sous-programme appelé.

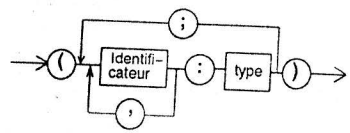
- Le paramètre RESULTAT, dont la valeur calculée dans le sous-programme appelé est transmise au sous-programme appelant.

- Le paramètre DONNEE/RESULTAT ou PAR VARIABLE, dont la valeur est modifiée dans le sous-programme appelant au moment même où elle est modifiée dans le sous-programme appelé.

■ PARAMETRES PAR VALEUR

La syntaxe de l'en-tête de la procédure appelée définit le type des paramètres par valeur.

Diagramme syntaxique paramètres valeur

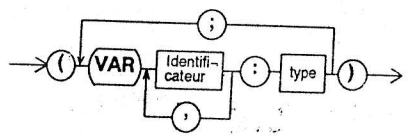


Le paramètre effectif est une expression qui est évaluée dans la procédure appelante. L'instruction d'appel donne au paramètre effectif la valeur résultant de cette évaluation ; cette valeur est transmise à la procédure appelée dans laquelle elle prend la place du paramètre formel. Le paramètre formel est toujours une variable locale à la procédure appelée.

■ PARAMETRES PAR VARIABLE

Dans la syntaxe de l'en-tête de la procédure appelée, le mot réservé **VAR** précède les paramètres formels. Les types des variables sont également précisés dans l'en-tête.

Diagramme syntaxique paramètres variable



Le paramètre effectif est ici une variable et non plus une expression. L'instruction d'appel mentionne les variables transmises à la procédure appelée, dans laquelle elles sont repérées par les paramètres formels.
Toute opération effectuée sur un paramètre formel agit instantanément sur le paramètre effectif correspondant. En effet, la procédure appelée travaille sur des variables formelles ayant les mêmes adresses que les variables effectives.

Les paramètres par variable portent aussi le nom de paramètres par référence, ou paramètres par adresse.

Les actions nommées de l'algorithme obéissent aux mêmes règles.

■ EXEMPLE NUMERIQUE SIMPLE

Pour bien comprendre et distinguer les modes de passage des paramètres par valeur et par variable, nous présentons ci-dessous un exemple de programme trivial, mais qui aura le mérite de la clarté.
L'objectif du programme suivant est d'utiliser un paramètre valeur, pour doubler une valeur, puis d'utiliser un paramètre variable pour tripler une variable, et enfin d'observer les résultats.

a) Algorithme

```
Début algorithme paramètre
   entier : A
Début action principale
   écrire 'Saisir un nombre entier'
   lire A
   écrire 'Nombre = ', A : 5
   écrire ''                          /* saut de ligne */
   doublevaleur(A)
   écrire 'Nombre = ', A : 5
   écrire ''                          /* saut de ligne */
   triplevariable(A)
   écrire 'Nombre = ', A : 5
Fin action principale

Action doublevaleur (B : entier)
Début action
   B <--- 2 * B
   écrire 'Valeur doublée = ', B : 5
Fin action

Action triplevariable (VAR C : entier)
Début action
   C <--- 3 * C
   écrire 'Variable triplée = ', C : 5
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM parametre (INPUT,OUTPUT);
VAR      A : INTEGER;

PROCEDURE doublevaleur (B : INTEGER);
BEGIN
   B := 2 * B;
   WRITELN('valeur doublee = ',B :5);
END;

PROCEDURE triplevariable
                (VAR C : INTEGER);
BEGIN
   C := 3 * C;
   WRITELN('variable triplee = ',C :5)
END;

BEGIN                    (* programme principal *)
   WRITELN('Saisir un nombre entier');
   READLN(A);
   WRITELN('Nombre = ',A :5);
   WRITELN;                        (* saut de ligne *)
   doublevaleur(A);
   WRITELN('Nombre = ',A :5);
   WRITELN;                        (* saut de ligne *)
   triplevariable(A);
   WRITELN('Nombre = ',A :5)
END.
```

c) Affichage écran

A l'exécution du programme, si l'utilisateur saisit la valeur 12, l'écran affichera :

```
Saisir un nombre entier
   12
Nombre =    12

valeur doublee =    24
Nombre =    12

variable triplee =    36
Nombre =    36
```

* Le nombre original est *A*, et l'utilisateur lui donne la valeur 12 : *Nombre = 12*.
* L'appel de la procédure *doublevaleur* passe à la procédure la valeur 12. Cette procédure reçoit la valeur dans la variable locale *B*, où *B* est doublée : *valeur doublee = 24*.
* La variable *A* est inchangée dans le programme principal qui écrit à nouveau : *Nombre = 12*.
* L'appel de la procédure *triplevariable* passe à la procédure l'adresse mémoire de la variable *A* dont la valeur est 12. Cette procédure reçoit cette valeur dans la variable locale *C*, de même adresse que la variable *A*, et *C* est triplée : *valeur triplee = 36*.
* Le contenu de l'adresse de la variable *A* a été modifié, et le programme principal écrit maintenant la nouvelle valeur de *A* : *Nombre = 36*.

Le programmeur veillera à la correspondance entre les paramètres effectifs de l'instruction d'appel et les paramètres formels de la procédure invoquée :

– Les nombre et place des paramètres devront bien être identiques dans les deux zones,
– Chaque paramètre effectif devra être de type compatible avec le paramètre formel correspondant.

■ EXEMPLE DE LA PROCEDURE ECHANGE.

La procédure echange utilisée dans les programmes ***ordonner3*** et ***ordonner4*** peut être redéfinie avec des paramètres par variable :

```
PROCEDURE echange(VAR x, y :REAL);
  VAR    tampon : REAL;
BEGIN
  tampon := x;
  x := y;
  y := tampon
END;
```

L'instruction d'appel du programme appelant est :

```
echange(petit,grand);
```

Remarque 1 : Lors de l'appel de la procédure paramétrée par variable *échange*, les paramètres effectifs *petit* et *grand* prennent la place des paramètres formels *x* et *y*. Ces derniers représentent des variables locales à *échange*, ayant respectivement les mêmes adresses que *petit* et *grand*. Toute modification de *x* et *y* se répercute instantanément sur *petit* et *grand*.

Remarque 2 : Une procédure paramétrée par variable, comme **echange(VAR x, y : REAL)**, présente plusieurs points d'intérêt :
– Une fois testée, elle réalisera des échanges certifiés sans erreur de deux variables réelles.
– Elle est écrite une seule fois pour toute dans un programme ayant à réaliser plusieurs échanges.
– Elle peut, si besoin, être recopier et utiliser dans un autre programme.

EXERCICE IV-3 : Ecrire une procédure paramétrée effectuant la permutation circulaire de 3 nombres A, B, C, saisis par l'utilisateur. Au terme du calcul, B contiendra la valeur origine de A, C celle de B, et A celle de C.

APPLICATION IV-4

Un robot d'assemblage reçoit une information sur la longueur de l'élément à manipuler. L'information est considérée comme valide si la longueur est comprise entre 30 et 125 ; dans ce cas, le processus continue, sinon il doit s'arrêter.
– Le robot actionne un moteur qui élève l'élément à une hauteur calculée en fonction de sa longueur (Elévation = 130 – Longueur), puis affiche la valeur de l'élévation,
– Il doit le maintenir en position pendant une durée proportionnelle à sa longueur (Durée de 500 unités de temps pour une longueur de 60).
– On donne quelques exemples de données et solutions du problème, illustré par la Figure B-18 :

Longueur	60	75	90	120
Elévation	70	55	40	10
Durée	500	625	750	1000

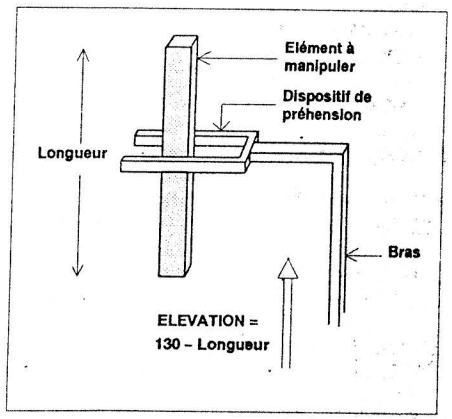


Fig. B-18. Robot d'assemblage

a) Analyse

L'algorithme et le programme suivant mettent en oeuvre cinq sous-programmes paramétrés :
– La procédure ***mesure*** permet la saisie de la longueur de l'élément à manipuler. Elle possède 2 paramètres par variable (*longueur* et *erreur*).
– La procédure ***monte*** réalise l'élévation de l'élément. La procédure ***calcule*** recherche et affiche la durée en position fixe. Ces 2 procédures possèdent chacune 1 paramètre par valeur et 1 paramètre par variable.
– Les procédures ***affiche*** *(elevation)* et ***attend*** *(duree)* possèdent un seul paramètre par valeur.

b) Algorithme

```
Début algorithme robot
  réel : longueur, elevation, duree
  logique : erreur
Début action principale
  mesure(longueur, erreur)
  si NON erreur
    alors
      monte(longueur,elevation)
      affiche(elevation)
      calcule(duree,longueur)
      attend(duree)
  Fin si
Fin action principale

Action mesure (VAR dim : réel,
               VAR faute : logique)
Début action
  écrire 'Mesure de la grandeur entre 30 et 125'
  lire dim
  si dim < 30 OU dim > 125
    alors
      faute <--- VRAI
      écrire 'Erreur lecture : Fin du Processus'
    sinon faute <--- FAUX
  Fin si
Fin action mesure

Action monte (dim : réel, VAR niveau :réel)
  logique :moteur /* VRAI=marche, FAUX=arrêt */
  réel : hauteur
Début action
  niveau <--- 130 - dim
  moteur <--- VRAI, hauteur <--- 0
  Tant que hauteur < niveau faire
    hauteur <--- hauteur + 1
  Fin Tant que
  moteur <--- FAUX
Fin action monte

Action affiche (niveau : réel)
Début action
  écrire 'Élévation égale a ', niveau :3:2
Fin action affiche

Action calcule (VAR delai : réel, taille : réel)
Début action
  delai <--- taille * 500 / 60
  écrire 'Attente durée de ', delai :3:2
Fin action calcule

Action attend (delai : réel)
Début action
  Répéter delai <---delai - 1 Jusqu'à delai <= 0
Fin action attend
Fin algorithme
```

c) Programme Pascal

```
PROGRAM robot (INPUT,OUTPUT);
VAR   longueur, elevation, duree : REAL;
      erreur : BOOLEAN;

PROCEDURE mesure(VAR dim : REAL;
                 VAR faute : BOOLEAN);
BEGIN
  WRITELN('Mesure grandeur entre 30 et 125');
  READLN(dim);
  IF (dim < 30) OR (dim > 125)
    THEN BEGIN
           faute := TRUE;
           WRITELN('Erreur : Fin Processus')
           END
    ELSE faute := FALSE
END;

PROCEDURE monte (dim : REAL;
                 VAR niveau : REAL);
VAR   moteur : BOOLEAN;
      (* VRAI = marche, FAUX = arrêt *)
      hauteur : REAL;
BEGIN
  niveau := 130 - dim;
  moteur := TRUE; hauteur := 0;
  WHILE hauteur < niveau DO
      hauteur := hauteur + 1;
  moteur := FALSE
END;
PROCEDURE affiche (niveau : REAL);
BEGIN
  WRITELN('Elevation egale a ', niveau :3:2 )
END;
PROCEDURE calcule (VAR delai : REAL;
                   taille : REAL);
BEGIN
  delai := taille * 500 / 60;
  WRITELN('Attente duree de ', delai :3:2 )
END;
PROCEDURE attend (delai : REAL);
BEGIN
  REPEAT delai := delai - 1 UNTIL delai <= 0
END;

BEGIN                    (* Programme principal *)
  mesure(longueur, erreur);
  IF NOT erreur
    THEN
      BEGIN
        monte(longueur,elevation);
        affiche(elevation);
        calcule(duree,longueur);
        attend(duree)
      END
END.
```


13. FONCTIONS

1°- DEFINITION ET DECLARATION

Nous avons déjà utilisé certaines fonctions standards telles que **SIN** (sinus), **SQR** (carré) ou **SQRT** (racine carrée). Les fonctions disponibles en Pascal, données à propos des expressions, n'utilisent qu'un seul paramètre. Les fonctions crées par le programmeur feront intervenir un ou plusieurs paramètres. On utilise souvent le terme **d'argument** pour désigner un paramètre de fonction.

Une fonction est un sous-programme chargé de calculer une valeur à partir d'une partie des données du programme. Contrairement à la procédure qui est appelée du programme principal au travers d'une instruction, la fonction est appelée à partir d'une expression du programme principal, dans laquelle elle retourne une valeur d'un type précisé dans l'en-tête du sous-programme appelé.

Notation algorithmique

```
Action nomaction(paramètres) : type
  partie des déclarations
  Début action
    ...
  Fin action
```

Syntaxe Pascal

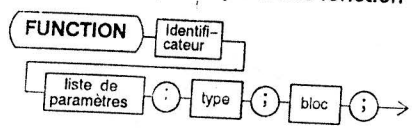
La structure d'une fonction est identique à celle d'une procédure ou d'un programme. Elle est constituée d'un en-tête et d'un bloc.

```
<fonction> ::= <en-tête> <bloc>
```

L'en-tête commence par le mot **FUNCTION** puis identifie la fonction par son identificateur, nom donné par le programmeur, et fournit une liste de paramètres ; L'en-tête précise enfin, le type de la valeur retournée par la fonction.

```
<en-tête de fonction> ::=
FUNCTION <identificateur>
        <liste de paramètres> :
          <type du résultat>
```

Diagramme syntaxique d'une fonction



- La fonction est activée, à partir du programme principal ou d'un sous-programme, lors de l'évaluation d'une expression contenant l'identificateur de la fonction, suivi des paramètres d'appel.

- Les paramètres d'appel doivent correspondre en nombre , en type, et en position aux paramètres formels de la fonction.

- La liste de paramètres peut être vide mais elle comporte généralement au moins un paramètre.

- Le bloc de la fonction est structuré comme la partie bloc d'une procédure ou d'un programme. On peut donc retrouver les parties :

  * déclaration d'étiquettes,
  * définition de constantes,
  * définition de types,
  * définition de variables,
  * procédures et fonctions
  * instructions exécutables.

La seule partie obligatoire du bloc est celle des instructions exécutables dans laquelle il y a obligatoirement une instruction d'affectation de l'identificateur de la fonction. Ce dernier doit impérativement prendre une valeur pour que la fonction puisse la retourner dans l'expression du sous-programme appelant.

Comme pour les procédures, les paramètres peuvent être passés par valeur ou par variable. Une fonction pourra aussi être passée comme paramètre d'une procédure ou d'une autre fonction. Nous verrons plus loin l'intérêt du paramètre fonction (analyse numérique et applications spéciales).

EXEMPLE NUMERIQUE SIMPLE

Fonction de calcul de la surface d'un disque.

a) Algorithme

```
Début algorithme disque
  réel : rayon
Début action principale
  écrire 'Saisir le rayon du disque'
  lire rayon
  écrire 'Surface = ', surface(rayon) :6:5
Fin action principale

Action surface(R: réel) : réel
  réel : pi = 3.14159
Début action
  surface <--- pi * R * R
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM disque (INPUT,OUTPUT);
VAR   rayon : REAL;

FUNCTION surface(R : REAL) : REAL;
  CONST  pi = 3.14159;
BEGIN
  surface := pi * R * R
END;

BEGIN                    (* Programme principal *)
  WRITELN('Saisir le rayon du disque');
  READLN(rayon);
  WRITELN('Surface = ', surface(rayon) :6:5);
END.
```

Remarque 1 : Lors de l'exécution de l'instruction d'écriture *'La surface est ',surface(rayon)*, le programme recherche la fonction *surface*, lui passe le paramètre effectif *rayon* , et le paramètre formel R prend la valeur de *rayon*. Le résultat de l'expression *pi * R * R* est ensuite affecté à la fonction *surface* (La constante *pi* est déclarée locale à la fonction *surface*).

Remarque 2 : Une fois testée, une fonction réalise des calculs certifiés sans erreur, elle peut être appelée plusieurs fois au cours d'un même programme ou , si besoin, être recopiée et utilisée dans un autre programme.

APPLICATION IV-5

Recherche de maximum dans une série de n mesures positives, saisies par l'utilisateur, en programmant une **fonction de calcul du maximum de 2 valeurs réelles.**

a) Algorithme

```
Début algorithme cherchemax
  entier : n , i
  réel : plusgrand, mesure
Début action principale
  plusgrand <--- 0
  écrire 'Saisir le nombre de mesures' , lire n
  Pour i de 1 à n faire
    écrire 'Saisir la mesure numéro ', i
    lire mesure
    plusgrand <--- maxi(plusgrand,mesure)
  Fin pour
  écrire 'Maxi des mesures =', plusgrand :4:3
Fin action principale

Action maxi(X,Y : réel) : réel
Début action
  si X > Y  alors maxi := X
            sinon maxi := Y
  Fin si
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM cherchemax (INPUT,OUTPUT);
VAR   n, i : INTEGER;
      plusgrand, mesure : REAL;

FUNCTION maxi(X,Y : REAL) : REAL;
BEGIN
  IF X > Y  THEN maxi := X
            ELSE maxi := Y
END;

BEGIN                    (* Programme principal*)
  plusgrand := 0;
  WRITELN('Saisir le nombre de mesures');
  READLN(n);
  FOR i := 1 TO n DO
    BEGIN
      WRITELN('Saisir la mesure numero ', i);
      READLN(mesure);
      plusgrand := maxi(plusgrand,mesure)
    END;
  WRITELN('Maxi des mesures ', plusgrand :4:3)
END.
```


APPLICATION IV-6

Une Entreprise calcule les salaires bruts de ses représentants en leur attribuant une partie fixe de 5000 F et une partie variable (commission) fonction du chiffre d'affaires (CA) réalisé :
+ 3 % pour la partie du chiffre d'affaires inférieur ou égale à 100 000 F et **+ 5%** pour la partie supérieure à ce montant (Fig. B-19, ci-contre).
Exemple : Pour le Chiffre d'Affaires de 120 000 F, le calcul de la partie variable est le suivant :
0,03 * 100 000 + 0,05 * 20 000 = 4000.

Calculer une série de salaires en incorporant une **fonction de calcul de la partie variable.** Le programme s'arrête lorsque l'utilisateur saisit un chiffre d'affaires négatif.

a) Algorithme

```
Début algorithme salaire
  entier : fixe = 5000,
  réel : salairebrut, CA,      /* Chiffre d'Affaires */
         plafond = 100000,     /* ou 0.1E+06 */
         taux1 = 0.03, taux2 = 0.05

Début action principale
  saisiCA                /* Saisie Chiffre Affaires */
  Tant que CA >= 0 faire
    salairebrut <--- fixe + commission(CA)
    écrire 'Salaire calculé = ', salairebrut :6:2
    saisiCA
  Fin Tant que
Fin action principale

Action saisiCA
Début action
  écrire 'Saisir le Chiffre d''Affaires (<0 = fin)'
  lire CA
Fin action

Action commission(montant : réel) : réel
Début action
  si montant > plafond
    alors
      commission <--- plafond * taux1 +
                      (montant - plafond) * taux2
    sinon commission <--- montant * taux1
  Fin si
Fin action
Fin algorithme
```

Remarque : Pour une modification des conditions de rémunération, il suffit de modifier les valeurs des constantes *fixe, taux1, taux2* et *plafond*.

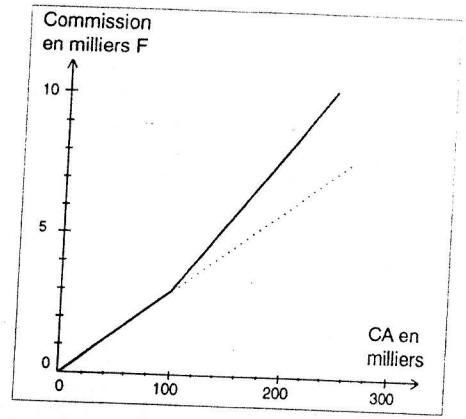


Fig. B-19. Commission en fonction du CA.

b) Programme Pascal

```
PROGRAM salaire(INPUT,OUTPUT);
CONST
      fixe = 5000; taux1 = 0.03; taux2 = 0.05;
      plafond = 100000;          (* ou 0.1E+06 *)
VAR   CA, salairebrut : REAL;

PROCEDURE saisiCA;
BEGIN
  WRITELN('Saisir le Chiffre d''Affaires (<0 =fin)');
  READLN(CA)
END;

FUNCTION commission(montant :real) : REAL;
BEGIN
  IF montant > plafond THEN
              commission := plafond * taux1 +
                (montant - plafond) * taux2
          ELSE commission := montant * taux1
END;

BEGIN                    (* Programme principal *)
  saisiCA;               (* Saisie Chiffre Affaires *)
  WHILE CA >= 0 DO
  BEGIN
    salairebrut := fixe + commission(CA);
    WRITELN('Salaire calcule=', salairebrut :6:2);
    saisiCA
  END
END.
```

EXERCICE IV-4 : Ecrire la Fonction TANG (paramètre x) calculant la tangente, rapport des fonctions SIN(x) et COS(x) intégrées au langage.

APPLICATION IV-7

Fonctions de calcul de DIV et MOD intégrées dans un programme de conversion d'un temps saisi en secondes par l'utilisateur, en heures, minutes et secondes.

a) Algorithme

```
Début algorithme seconde2
   entier : temps
Début action principale
   écrire 'saisir un temps en secondes'
   lire temps
   écrire divi(temps,3600) :2,' heures  '
   écrire divi(modu(temps,3600),60) :2
   écrire ' minutes  ')
   écrire modu(modu(temps,3600),60) :2
   écrire ' secondes')
Fin action principale

Action divi(a,b : entier) :entier
Début action                          /* fonction divi */
   divi := a DIV b
Fin action
Action modu(a,b : entier) :entier
Début action                          /*fonction modu*/
   modu := a MOD b
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM seconde2(INPUT,OUTPUT);
VAR    temps : INTEGER;

FUNCTION divi(a,b : INTEGER) : INTEGER;
BEGIN
  divi := a DIV b;
END;

FUNCTION modu(a,b : INTEGER) : INTEGER;
BEGIN
  modu := a MOD b;
END;

BEGIN                    (* programme principal*)
  WRITELN('saisir un temps en secondes');
  READLN(temps);
  WRITE( divi(temps,3600) :2,' heures  ');
  WRITE( divi(modu(temps,3600),60) :2);
  WRITE(' minutes  ');
  WRITE( modu(modu(temps,3600),60) :2);
  WRITELN(' secondes')
END.
```

Remarque 1 : Le programme principal demande l'écriture de *divi(temps,3600)* formatée sur 2 positions. La fonction *divi* est appelée avec les paramètres effectifs *temps* et *3600*.

Remarque 2 : La fonction *divi* comme la fonction *modu* reçoit 2 paramètres entiers qui viennent se substituer à *a* et *b* ; *divi* comme *modu* vont ensuite calculer à partir des paramètres formels *a* et *b*. Les valeurs de *divi* et *modu* sont enfin retournées dans l'expression d'appel.

2°- LE PARAMETRE FONCTION

Une fonction ou une procédure peut être définie avec un paramètre fonction. L'exemple numérique suivant met en oeuvre une fonction possédant un paramètre fonction.

■ EXEMPLE NUMERIQUE SIMPLE

Programme de calcul de la somme des carrés et de la somme des racines carrées des 10 premiers nombres entiers.

Programme Pascal

```
PROGRAM paramfonction(INPUT,OUTPUT);
CONST  debut = 1; fin = 10;

Function S(Function F :REAL;
                X,Y :INTEGER) :REAL;
VAR      i : INTEGER;
         som : REAL;
BEGIN
  som := 0;
  FOR i := X TO Y DO som := som + F(i);
  S := som
END;

BEGIN                    (* Programme principal*)
  WRITELN('Som carres = ', S(SQR,debut,fin);
  WRITELN('Som racines = ', S(SQRT,debut,fin)
END.
```

Remarque 1 : La fonction de calcul *S*, incorpore dans ses paramètres la fonction *F*, qui aura pour valeur **SQR lors du premier appel et SQRT lors du second appel.**

Remarque 2 : Certaines versions de Pascal n'autorisent pas l'utilisation des paramètres fonction et procédure.

14. RECURSIVITE

■ DEFINITION

La récursivité donne la possibilité de faire figurer dans la définition d'un objet une référence à ce même objet. Par exemple, la somme des n premiers entiers peut faire référence à la somme des (n-1) premiers entiers, en écrivant :

Som(n) = n + Som(n-1) avec **Som(1) = 1**

Une procédure ou une fonction récursive est un sous-programme qui s'appelle lui-même, et qui intègre une variable de contrôle permettant d'arrêter le processus d'appel.

■ EXEMPLE NUMERIQUE SIMPLE

Calcul de la somme des n premiers entiers (S = 1 + 2 + 3 +...n) en utilisant une méthode classique puis une méthode récursive.

a) METHODE CLASSIQUE : Algorithme

```
Début algorithme somme1
   entier : n, i, som
Début action
   écrire 'Saisir le nombre entier'
   lire n
   som <--- 0
   pour i de 1 à n faire
      som <--- som + i
   Fin pour
   écrire 'Somme nombres = ',som
Fin action
Fin algorithme
```

b) METHODE CLASSIQUE : Programme

```
PROGRAM somme1 (INPUT,OUTPUT);
VAR      n, i, som : INTEGER;
BEGIN
   WRITELN('Saisir le nombre entier');
   READLN(n);
   som := 0;
   FOR i := 1 TO n DO som := som + i;
   WRITELN('Somme nombres = ', som :5)
END.
```

c) METHODE RECURSIVE

1- Analyse

S(n) = 1 pour n = 1
S(n) = n + S(n-1) pour n > 1

2- Programme

```
PROGRAM somme2 (INPUT,OUTPUT);
VAR          n, i : INTEGER;

FUNCTION som (val : INTEGER) : INTEGER;
BEGIN
  IF val = 1
    THEN som := 1
    ELSE som := val + som(val - 1)
END;

BEGIN                    (* Programme principal *)
  WRITELN('Saisir le nombre entier');
  READLN(n);
  WRITELN('Somme nombres = ', som(n) :5);
END.
```

Remarque : Si l'on remplace les signes + par des signes * , on calcule F = 1 * 2 * 3 *...n, qui représente factorielle de n. L'analyse du problème récursif serait la suivante :

F(n) = 1 pour n = 1
F(n) = n * F(n-1) pour n > 1.

■ APPLICATIONS ET LIMITES DE LA RECURSIVITE

L'exemple précédent est trivial au point que le programme n'utilisant pas la méthode récursive est plus simple. Mais d'autres calculs sont fort utiles, comme celui de X puissance n, qui peut s'écrire : $X^n = X * X^{n-1}$ avec $X^0 = 1$

La récursivité trouve tout son intérêt et démontre sa puissance, lorsque la solution d'un problème complexe est écrite en faisant référence à la solution du même problème pris dans un cas plus simple.

Les programmes récursifs sont souvent "gourmands" en mémoire car il génèrent autant de variables locales qu'il y a de niveaux d'appel. Ces appels à la fonction récursive doivent être empilés en mémoire, pour être dépilés en fin de processus.

V - STRUCTURES DE DONNEES

■ INTRODUCTION

Tous les algorithmes et programmes présentés jusqu'ici ont manipulé des variables simples ; à un instant donné, la variable ne possédait qu'une valeur. Certains problèmes nécessitent l'utilisation de **variables structurées possédant simultanément plusieurs valeurs.**

On peut par exemple vouloir représenter une variable *point*, telle qu'elle contienne 2 valeurs réelles, l'une pour la partie abscisse et l'autre pour la partie ordonnée.
On peut également avoir besoin de travailler avec une variable indicée, de représenter des vecteurs ou des matrices.

Ces notions vont nous conduire à la définition de variables contenant une liste de valeurs de même type. Ces listes pourront contenir des valeurs numériques, logiques ou des caractères. Nous pourront ainsi créer et manipuler des chaînes de caractères. Ces différentes structures de données sont représentées par le **TYPE TABLEAU.**

Nous verrons également dans ce chapitre que le langage Pascal permet la définition de **TYPE ENSEMBLE**, au sens mathématique du terme, rendant ainsi possible l'utilisation des opérateurs classiques de la Théorie des Ensembles.

Nous aurons besoin de variables permettant, pour une même entité, de lui rattacher plusieurs valeurs de types différents.
Par exemple, à un produit, on rattache une référence (type entier) identifiant le produit, un libellé (type chaîne de caractères) et un prix (type réel). Une variable ainsi structurée se réfère au **TYPE ENREGISTREMENT.**

La suite des valeurs d'une variable de type ENREGISTREMENT, peut être regroupée au sein d'un **FICHIER.** En effet un Fichier présente la particularité de stocker des données de type CARACTERE ou de type ENREGISTREMENT. Ce stockage peut faire appel à un support externe de l'ordinateur (disque, disquette ...).

Le type **POINTEUR** permet de manipuler des variables dynamiques, créées pendant l'exécution du programme, en quantité correspondant à un besoin non chiffrable à priori. Une variable de type POINTEUR contient une adresse.

15. VECTEUR OU TABLEAU A UNE DIMENSION

Une tableau est un ensemble de zones contigües contenant des valeurs de même type. Chaque zone est repérée et identifiée par un ou plusieurs indices.

Lorsque l'indice est unique, le tableau est dit à **UNE DIMENSION.** On l'appelle aussi **VECTEUR.**
Lorsque N indices sont nécessaires pour repérer une valeur le tableau est dit à **N DIMENSIONS.**

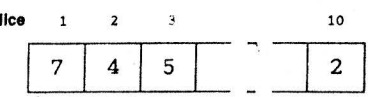
La structure de tableau à une dimension est très répandue. Elle est utilisée dans les files, les listes, elle permet la représentation des chaînes de caractères, de toute forme de vecteurs, ou la simulation du fonctionnement d'une pile.

Bien que les tableaux à 1 et N Dimensions présentent des formes de déclarations identiques, nous séparons les chapitres les présentant, tant leurs applications sont nombreuses et variées.

Considérons une série de 10 objets dont une grandeur est mesurée (10 notes d'élèves, ou 10 poids de colis ...), et pour lesquels on souhaite conserver en mémoire le numéro de l'objet ainsi que la grandeur qui lui est affectée.

Cette liste de mesures peut être représentée sous la forme mathématique d'une variable indicée:
N_i pour des notes
P_i pour des poids
avec l'indice i variant de 1 à 10.

Les langages informatiques évolués présentent ce type de variable à l'aide d'un Tableau à 1 Dimension ou VECTEUR, contenant 10 cases contigües. Chaque case est repérée par un indice, pouvant varier de 1 à 10, et contient la valeur mesurée de l'objet considéré.



Le vecteur schématisé ci-dessus comporte 10 éléments du type entier.

Le programmeur donne un nom au Vecteur ; ce nom identifie la structure. Dans le cas des notes, il peut le nommer ***noteeleve***. Dans le programme, on accèdera à la note de l'élève numéro 3 par la notation ***noteeleve[3]***, et plus généralement à la note de l'élève numéro i par la notation ***noteeleve[i]***.

Notation algorithmique de vecteur

type éléments : nomtableau [type indice]

- *Type éléments* est le type des éléments du vecteur, soit un type simple (entier, réel, caractère ou logique), soit un type structuré autre que tableau.
- *Nomtableau* est l'identificateur de la structure.
- *Type indice* est un type scalaire (énuméré), ou un type intervalle dans lequel l'indice prendra ses valeurs.

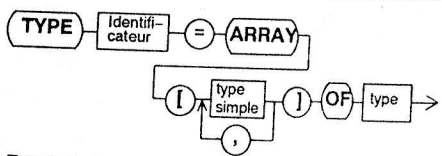
Exemple :

réel : Noteeleve[1..10]

déclare un vecteur permettant le stockage de 10 nombres réels. Chaque nombre est repéré par un indice entier compris entre 1 et 10.

Syntaxe Pascal et Diagramme Syntaxique

<type tableau> ::=
ARRAY [<type indice>] **OF** <type élément>



Exemples :

Noteeleve = **ARRAY [1..10] OF REAL;**
Valeur = **ARRAY ['A'..'Z'] OF INTEGER;**
Caract = **ARRAY [0..12] OF CHAR;**
chiffre = 0..5;
Resultat = **ARRAY[chiffre] OF BOOLEAN;**

Remarque : Le nombre des éléments d'un vecteur est déterminé lors de sa déclaration. On connaît donc exactement le nombre de lectures à effectuer pour réaliser le remplissage de ce vecteur. **On retiendra que la structure répétitive POUR est parfaitement adaptée pour les opérations simples sur les vecteurs.**

■ REMPLISSAGE D'UN VECTEUR

Nous travaillerons sur un exemple de **remplissage, ou lecture** d'un vecteur :
Algorithme et Programme enregistrant dans un vecteur *poids*, les poids de 10 colis, saisis par l'utilisateur. Calculer le poids total.

a) Algorithme

```
Début algorithme colis
  entier : i                          /* indice du vecteur */
  réel : poids [1..10], total
Début action
  total <--- 0
  Pour i de 1 à 10 faire
    écrire 'Saisir poids du colis numéro', i
    lire poids[ i ]
    total <--- total + poids[ i ]
  Fin Pour
  écrire 'Le poids total est : ', total :5:2
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM colis (INPUT, OUTPUT);
    (* 1ERE FORME DE DECLARATION *)
  CONST  maxi = 10;
  TYPE   tabpoids = ARRAY[1..maxi] OF REAL;
  VAR    poids : tabpoids;
         i : INTEGER;          (* indice du vecteur *)
         total : REAL;

    (* 2EME FORME DE DECLARATION *)
(* TYPE  tabpoids = ARRAY[1..10] OF REAL; *)
(* VAR   poids : tabpoids; *)
(*       i : INTEGER; *)
(*       total : REAL; *)

    (* 3EME FORME DE DECLARATION *)
(* VAR   i : INTEGER;  total : REAL;*)
(*       poids : ARRAY[1..10] OF REAL; *)

BEGIN
  total := 0;
  FOR i := 1 TO maxi DO
    BEGIN
      WRITELN('Poids colis numero', i);
      READLN(poids[ i ]);
      total := total + poids[ i ]
    END;
  WRITELN('Le poids total est : ', total :5:2);
END.
```

Remarque 1 : Les trois possibilités de déclaration du vecteur *poids* sont équivalentes, mais l'une ou l'autre sera retenue en fonction des conditions d'utilisation.

– La 1ère forme, très complète, définit d'abord la constante *maxi* égale à 10, puis le type *tabpoids* et enfin le vecteur *poids* du type *tabpoids*.
– La 2ème forme est identique à la première à ceci près que l'intervalle de définition du type *tabpoids* est donné directement de 1 à 10.
– La 3ème forme définit directement le vecteur au niveau de la variable *poids*. Cette 3ème forme est plus concise, mais nécessite, dans le cas où plusieurs variables sont à créer avec le même type vecteur, de réécrire la définition du vecteur au niveau de chacune d'entre elles.

Par ailleurs, l'intérêt de définir la constante *maxi* réside dans le fait que lorsque sa valeur est à modifier, il suffit de le faire une seule fois, sur la ligne CONST, plutôt que de rechercher dans un long programme les occurrences de la valeur à modifier.

Remarque 2 : Comme pour toute variable, deux méthodes sont possibles pour donner une valeur à un élément de vecteur.

– Soit la lecture :

lire poids[3]	READLN(poids[3]);
Ou si i =3	
lire poids[i]	READLN(poids[i]);

– Soit l'assignation (affectation) :

poids[3] <--- 2.5	poids[3] := 2.5;
Ou si i =3	
poids[i] <--- 2.5	poids[i] := 2.5;

■ ECRITURE D'UN VECTEUR

Connaissant le nombre d'éléments du vecteur, l'écriture des éléments, comme la lecture, met en oeuvre la structure répétitive POUR.

```
Pour i de 1 à maxi faire
   écrire poids[ i ]
Fin Pour
```

```
FOR i := 1 TO maxi DO WRITELN(poids[ i ]);
```

■ TRI D'UN VECTEUR

Les opérations de tri constituent une part importante des traitements informatiques. Un tri consiste **à classer un ensemble d'éléments selon une relation d'ordre.**

Il existe plusieurs algorithmes de tri sur les vecteurs, mais la simplicité n'est pas synonyme de performance (durée nécessaire au traitement). L'algorithme suivant bien que peu performant est intéressant sur le plan méthodologique. Il met en oeuvre la procédure paramétrée *échange* que nous avons créée à la page 59.

APPLICATION V-1

Trier un Vecteur par la Méthode d'Extraction.
Le vecteur de dimension n=5, contient dans l'ordre les réels suivants : 7.2 3.6 9.1 2.3 5.4

a) Principe

– On compare le 1er élément avec tous les autres (n–1 comparaisons). A chaque comparaison, si l'élément comparé est inférieur à l'élément en 1ère position, on l'échange avec l'élément de 1ère position.
– On reproduit cette opération à partir du 2ème élément, puis du 3ème, jusqu'à partir du n–1ème.
– La ième opération consiste donc à comparer le ième élément avec le i+1ème, jusqu'au Nème, et à échanger à chaque fois le plus petit trouvé avec l'élément de position i.

b) Algorithme

```
Début algorithme tri1
   entier : i, n = 5          /* i = indice du vecteur */
   réel :   val[1..n]         /* vecteur de 5 réels */
Début action principale
   /* initialisation du vecteur */
   val[1] <--- 7.2, val[2] <--- 3.6, val[3] <---9.1,
   val[4] <--- 2.3, val[5] <--- 5.4
   Pour i de 1 à n-1 faire
      classement
   Fin Pour
   Pour i de 1 à n faire
      écrire val[ i ] :3:1
   Fin Pour
Fin action principale
```



```
Action classement
   entier : j
Début action
   Pour j de i+1 à n faire
      si val[ j ] < val[ i ]
         alors echange(val[ j ],val[ i ])
      Fin si
   Fin Pour
Fin action

Action echange(VAR x,y : ENTIER)
   entier : tampon
Début action
   tampon <--- x
   x <--- y
   y <--- tampon
Fin action
Fin algorithme
```

c) Programme Pascal

```
PROGRAM tri1 (INPUT,OUTPUT);
CONST   n = 5;          (* dimension du vecteur *)
VAR     i : INTEGER;    (* i = indice du vecteur*)
        val : ARRAY[1..n] OF REAL;

PROCEDURE echange(VAR x, y :REAL);
VAR   tampon : REAL;
BEGIN
  tampon := x;
  x := y;
  y := tampon
END;

PROCEDURE classement;
VAR   j : INTEGER;
BEGIN
  FOR j := i+1 TO n DO
    IF val[ j ] < val [ i ]
      THEN echange(val[ j ],val[ i ])
END;

BEGIN                    (* Programme principal *)
  (* initialisation du vecteur *)
  val[1] := 7.2;  val[2] := 3.6;  val[3] := 9.1;
  val[4] := 2.3;  val[5] := 5.4;
  FOR i := 1 TO n-1 DO classement;
  FOR i := 1 TO n DO WRITELN(val[ i ] :3:1)
END.
```

Remarque : La procédure appelée *echange* est placée dans le programme **avant** la procédure appelante *classement* (dont l'instruction unique commence par *FOR* et se termine par *val[i])*, elle même placée **avant** et **appelée n fois** par la boucle FOR du **programme principal**.

d) Trace de Tri1

i	val[i]	j	val[j]	échange	Instant
Etat initial du Vecteur : 7.2 3.6 9.1 2.3 5.4					
Premier passage (i = 1)					
1	**7.2**	2	**3.6**	oui	avant
1	**3.6**	2	7.2		après
1	3.6	3	9.1	non	
1	**3.6**	4	**2.3**		avant
1	**2.3**	4	**3.6**		après
1	2.3	5	5.4	non	
Etat du Vecteur: **2.3** 7.2 9.1 3.6 5.4					
Deuxième passage (i = 2)					
2	7.2	3	9.1	non	
2	**7.2**	4	**3.6**	oui	avant
2	**3.6**	4	**7.2**		après
2	3.6	5	5.4	non	
Etat du Vecteur: **2.3 3.6** 9.1 7.2 5.4					
Troisième passage (i = 3)					
3	**9.1**	4	7.2	oui	avant
3	**7.2**	4	**9.1**		après
3	**7.2**	5	**5.4**	oui	avant
3	**5.4**	5	7.2		après
Etat du Vecteur: **2.3 3.6 5.4** 9.1 7.2					
Quatrième passage (i = 4)					
4	**9.1**	5	7.2	oui	avant
4	**7.2**	5	**9.1**		après
Etat final du Vecteur : 2.3 3.6 5.4 7.2 9.1					

Remarque : Il existe de nombreux algorithmes de tri dont certains peuvent être très complexes et très performants. Pour d'autres, la performance varie en fonction de l'ordre original des éléments à trier, mais la performance est toujours une fonction du nombre des éléments à trier.
Notre objectif n'est pas d'approfondir le domaine étendu des tris, traité dans des ouvrages spécialisés. Nous nous limiterons à citer les algorithmes les plus connus :
– Méthode de sélection de minimum,
– Méthode de tri insertion,
– Méthodes dites "par remontée des bulles",
– Méthode récursive par partition,
– Méthodes avec listes chaînées, arbres ...

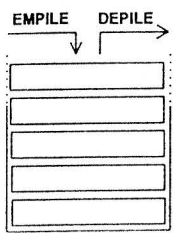
EXERCICE V-1 : Etant donné 3 vecteurs d'entiers X, N, P, de 10 éléments chacun, remplir par lecture le vecteur X, puis copier respectivement dans P et dans N, les valeurs Positives et Négatives ou Nulles de X. Afficher enfin les contenus de X, P, N.

■ SIMULATION DE PILE

La structure de vecteur permet la représentation d'ensembles de données pour lesquels existe un ordre de traitement des différents éléments.
Ainsi, dans le cas d'une **FILE**, seul l'élément de tête peut être traité, tandis que l'élément de queue ne sera traité (si nécessaire) qu'en dernier, après tous les autres. On dit que la File fonctionne selon le principe **FIFO (First In, First Out)**, autrement dit, **le premier élément entré est le premier sorti** (Cf Fig B-5 représentant la File comme un tunnel).

Une PILE est un ensemble de données fonctionnant selon le principe LIFO (Last In, First Out), tel que le dernier élément entré est le premier sorti.

On peut en donner de nombreux exemples : Pile de cartes dans un sabot, pile d'assiettes, Pile de copies à corriger, ... En Informatique, la structure de pile est très utilisée pour la sauvegarde temporaire d'informations. On la retrouve pour gérer les blocs d'un programme Pascal : chaque END correspond au dernier BEGIN rencontré.



— Le **SOMMET** est le dernier élément ajouté dans la Pile. C'est le seul élément pouvant être consulté ou supprimé.
— L'élément de **BASE** est le plus ancien de la Pile.

APPLICATION V-2

Ecrira l'algorithme et le programme permettant de **gérer l'ajout, la suppression et la consultation de caractères dans une Pile** ayant une capacité maximale de 10 éléments. On créera les procédures EMPILE, DEPILE et CONSULTE.
En fin de programme, une procédure devra lister le contenu de la pile.

a) Analyse

— Soit une Pile contenant des caractères, et représentée par un Vecteur, contenant au maximum le nombre entier *dimpile* caractères.

— Appelons *sommet* l'indice de ce Vecteur. Il doit tenir compte du fait que la Pile peut être vide *(sommet = 0)*, et que la Pile peut être pleine *(sommet = dimpile)*.

— **AJOUTER** un élément dans la Pile correspond à l'action **EMPILE**. Cette action n'est possible que si la Pile n'est pas pleine. Avant l'ajout, l'indice *sommet* sera incrémenté de 1 unité.

— **SUPPRIMER** un élément de la Pile correspond à l'action **DEPILE**. Cette action n'est possible que si la Pile n'est pas vide. Après la suppression, l'indice *sommet* sera décrémenté de 1 unité.

— **CONSULTE** est l'action qui permet d'afficher l'élément positionné au sommet de la Pile.

— **Résumons les déclarations nécessaires :**

dimpile	est une constante entière,
sommet	est une variable entière (indice)
pile	est un Vecteur de *dimpile* caractères.
element	est une variable de type caractère.

b) Algorithme

```
Début algorithme Gestion de Pile
  entier :      sommet, dimpile = 10,
  caractère :   element, choix
                Pile[1..dimpile]

Début action principale
  sommet <--- 0                          /* Pile vide */
  Répéter
    écrire 'Choisir une option (Majuscule)'
    écrire 'Empiler = E, Dépiler = D,'
    écrire 'Consulter sommet = C, Terminer = T'
    lire choix
    Selon que choix
      'E' : empile
      'D' : depile
      'C' : consulte
    Fin Selon
  Jusqu'à choix = 'T'
  listepile
Fin action principale
```



```
Action empile
Début action
  si sommet = dimpile          /* La Pile est pleine */
    alors écrire 'Impossible : Pile pleine'
    sinon
      écrire 'Saisir le caractère a empiler'
      lire element
      sommet <--- sommet + 1
      Pile[sommet] <--- element
  Fin si
Fin action

Action depile
Début action
  si sommet = 0                /* La Pile est vide */
    alors écrire 'Impossible : Pile vide'
    sinon
      element <--- Pile[sommet]
      sommet <--- sommet - 1
      écrire 'caractère dépilé : ', element
  Fin si
Fin action

Action consulte
Début action
  si sommet = 0
    alors écrire 'Impossible : Pile vide'
    sinon écrire 'Le sommet est : ',Pile[sommet]
  Fin si
Fin action

Action listepile
  entier : i
Début action
  écrire                       /* saut de ligne */
  si sommet = 0
    alors écrire 'Pile vide'
    sinon
      Pour i de 1 a sommet faire
        écrire Pile[ i ]
      Fin Pour
  Fin si
Fin action

Fin algorithme
```

c) Programme Pascal

```
PROGRAM pile (INPUT,OUTPUT);
CONST
    dimpile = 10;
VAR
    sommet : INTEGER;
    element, choix : CHAR;
    pile : ARRAY[1..dimpile] OF CHAR;

PROCEDURE empile;
BEGIN
  IF sommet = dimpile
    THEN WRITELN('Impossible : Pile pleine')
    ELSE
      BEGIN
        WRITE('Saisir le caractere a empiler ');
        READLN(element);
        sommet := sommet + 1;
        pile[sommet] := element
      END
END;

PROCEDURE depile;
BEGIN
  IF sommet = 0
    THEN WRITELN('Impossible : Pile vide')
    ELSE
      BEGIN
        element := pile[sommet];
        sommet := sommet - 1;
        WRITELN('caractere depile : ', element)
      END
END;

PROCEDURE consulte;
BEGIN
  IF sommet = 0
    THEN WRITELN('Impossible : Pile vide')
    ELSE
    WRITELN('Le sommet est : ',pile[sommet])
END;

PROCEDURE listepile;
  VAR   i : INTEGER;
BEGIN
  WRITELN;                     (* saut de ligne *)
  IF sommet = 0
    THEN WRITELN('Pile vide')
    ELSE FOR i := 1 TO sommet DO
           WRITELN(pile[ i ])
END;

BEGIN                          (* Programme principal *)
  sommet := 0;                 (* Pile vide *)
  REPEAT
    WRITELN('Choisir une option (Majuscule)');
    WRITELN('Empiler = E, Depiler = D,');
    WRITELN('Consulte = C, Termine = T');
    READLN(choix);
    CASE choix OF
      'E' : empile;
      'D' : depile;
      'C' : consulte
    END
  UNTIL choix = 'T';
  listepile
END.
```

d) Fonctionnement de EMPILE et de DEPILE

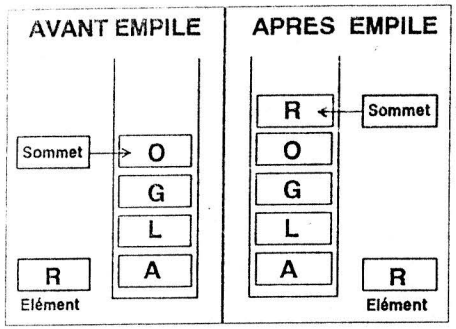


Fig. B-20. Fonctionnement de EMPILE

La procédure EMPILE réalise, dans l'ordre, les actions suivantes :
- Elle vérifie que la Pile n'est pas pleine, c'est à dire que l'indice *sommet* est différent de la dimension *dimpile* du Vecteur *Pile*,
- Elle demande à l'utilisateur de saisir le caractère à empiler (soit ***A, L, G, O, R***),
- Elle incrémente l'indice *sommet* de une unité, et enfin intègre le nouveau caractère à la position *sommet* du Vecteur *Pile*.

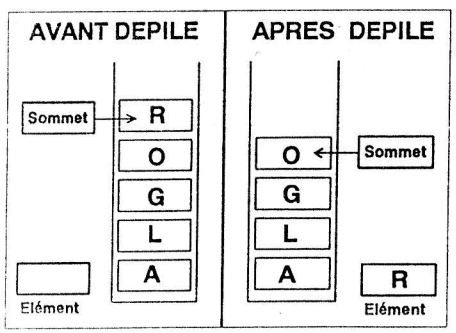


Fig. B-21. Fonctionnement de DEPILE

La procédure DEPILE réalise, dans l'ordre, les actions suivantes :
- Elle vérifie que la Pile n'est pas vide, c'est à dire que l'indice *sommet* est différent de zéro,
- Elle donne à la variable *element* la valeur du caractère situé au sommet de la Pile,
- Elle décrémente l'indice *sommet* de une unité.

CHAINES DE CARACTERES

Jusqu'à présent, nous n'avons utilisé que des variables caractères, correspondant au type CHAR, et qui, par définition, ne contiennent qu'un seul caractère. Ceci est bien sûr limité car il est souvent nécessaire de manipuler des chaînes de caractères.

Des applications devront par exemple utiliser des variables telles que :
- Le nom d'une personne : 'DUPONT'
- Un nom de Ville : 'NANTES'
- Le libellé d'un produit : 'Machine à laver'
- Un message : 'Cuve Pleine' ...

1- DECLARATION

Certaines versions de Pascal et notamment le Pascal UCSD ainsi que le Turbo Pascal, permettent de déclarer directement les chaînes de caractères en utilisant le type STRING. L'algorithmique procède de la même logique.

Algorithmique
chaine de 20 caractères : libelle
Pascal UCSD et TURBO Pascal
VAR **libelle : STRING(20);**

MAIS, le Pascal standard, ne disposant pas du type STRING, doit faire appel à un VECTEUR pour manipuler les chaînes de caractères.

Les exercices précédents sur les vecteurs ont montré que l'on accède séparément à tel ou tel élément du vecteur. L'objectif est maintenant de manipuler globalement une chaîne de caractères contenue dans un vecteur.

Le Pascal standard permet la déclaration de structures compactées par l'utilisation du mot réservé PACKED.

Exemples :
```
VAR
   nom :   PACKED ARRAY [1..15] OF CHAR;
   libelle : PACKED ARRAY [1..20] OF CHAR;
```

Le mot réservé PACKED provoque le compactage des caractères pour optimiser la place mémoire requise pour leur stockage.
L'indice est de type intervalle d'entiers dont la borne inférieure est égale à 1.

La variable *nom* contient 15 caractères. Les valeurs de *nom* devront toujours comporter 15 caractères. La valeur 'DUPONT' est donc insuffisante puisqu'elle ne contient que 6 caractères. Il faudra lui donner la valeur :
'DUPONT ', soit
DUPONT suivi de 9 caractères espace (blancs).
La variable *libelle* contient 20 caractères. La valeur 'Machine à laver' de cette variable devra être enregistrée 'Machine à laver ', incorporant 5 caractères blancs.

2- LECTURE ET ECRITURE D'UNE CHAINE AVEC LE TYPE STRING

Le Pascal standard n'accepte pas le type STRING, mais presque toutes les versions récentes du langage intègrent ce type STRING, prédéfinissant ainsi les chaînes de caractères. Nous en donnons ci-dessous un exemple d'utilisation.

Définir une variable NomEleve, chaîne de 15 caractères, puis lire et écrire cette variable.

a) Algorithme

```
Début algorithme chaine1   /* Pour Turbo Pascal*/
                           /*  ou Pascal UCSD */
   chaine de 15 caractères : NomEleve
Début action
   écrire 'Saisir le nom de l''élève'
   lire NomEleve
   écrire 'Le nom saisi est : ', NomEleve
Fin action
Fin algorithme
```

b) Programme Pascal UCSD et TURBO Pascal

```
PROGRAM chaine1(INPUT,OUTPUT);
                          (*Pour Turbo Pascal*)
                          (*  ou Pascal UCSD *)
CONST
      nbcar = 15;
TYPE
      Nom = STRING[nbcar];
VAR
      NomEleve : Nom;
BEGIN
   WRITELN('Saisir le Nom de l''eleve');
   READLN(NomEleve);
   WRITELN('Le nom saisi est : ', NomEleve)
END.
```

3- LECTURE ET ECRITURE D'UNE CHAINE EN PASCAL STANDARD

La saisie d'une chaîne de caractères est réalisée dans un Vecteur, caractère par caractère, à l'aide de l'instruction READ, jusqu'à ce que l'opérateur termine l'opération de lecture en appuyant sur la touche **Entrée** du clavier (ou **Retour Chariot**). L'action de cette touche provoque le passage de **la fonction booléenne EOLN de FAUX à VRAI** (EOLN signifiant End Of Line, c'est à dire Fin de Ligne).

Le programme suivant met en oeuvre une procédure paramétrée que nous appelons *saisichaine*. Cette procédure complète, si nécessaire, la saisie de l'opérateur par des caractères blancs, insérés en fin de vecteur.

Programme Pascal Standard

```
PROGRAM chaine2(INPUT,OUTPUT);
CONST
   nbcar = 20;
TYPE
   chaine = PACKED ARRAY[1..nbcar] OF CHAR;
VAR
   libelle1, libelle2 : chaine;

PROCEDURE saisichaine
        (longueur: INTEGER; VAR ch:chaine);
VAR    i,j : INTEGER;
       car : CHAR;
BEGIN
   WRITE('Saisir la Chaine de caracteres ');
   WRITELN('et terminer par la touche ENTREE');
   i := 1;
   REPEAT
      READ(car);
      ch[ i ] := car;
      i := i + 1
   UNTIL EOLN OR (i > longueur);
   WRITELN;
   FOR j := i TO longueur DO ch[ j ] := ' '
END;                        (* 1 espace entre ' et ' *)

BEGIN                       (* Programme principal *)
   saisichaine(nbcar,libelle1);
   WRITELN(libelle1);
   libelle2 := 'chaine de 20 maillon';
   WRITELN(libelle2)
END.
```

Remarque 1 : La variable *libelle2* reçoit une valeur par affectation directe, tandis que la

variable *libelle1* fait l'objet d'une lecture par la procédure *saisichaine*. Cette dernière possède 1 paramètre par valeur et 1 paramètre par variable.

Remarque 2 : La lecture (saisie) de la chaine *libelle1* ne peut pas être réalisée par l'instruction :

READ (libelle1);

alors que son écriture est tout à fait autorisée par :

WRITE(libelle1) ou WRITELN(libelle1);

Notons que WRITE(libelle1) est équivalent à :

FOR I := 1 TO nbcar DO WRITE(libelle1[I]);

4- OPERATIONS SUR LES CHAINES

Les opérateurs relationnels s'appliquent aux chaînes de caractères. On peut ainsi tester l'égalité ou l'inégalité de 2 chaînes de même longueur. On peut également utiliser les opérateurs **> >= < <=**.

Exemple : 'CLAUDE' est inférieur à 'DANIEL'
'DANIEL' est inférieur à 'DAMIEN'

5- PROCEDURES PACK ET UNPACK

Un vecteur chaîne de caractères compactés est déclaré par **PACKED ARRAY**. Du fait du compactage, le temps d'accès à un élément du vecteur est plus long. Si nécessaire, le vecteur peut être décompacté par la **procédure UNPACK**, puis recompacté, après le traitement, par la **procédure PACK**.

Exemple : Si TA : ARRAY[1..15] OF CHAR
et TB : PACKED ARRAY[1..15] OF char
PACK(TA,1,TB), et **UNPACK(TB,1,TA)**, réalisent à partir du 1er élément, le compactage de TA dans TB, et le décompactage de TB dans TA .

■ GESTION DES MESSAGES

Certaines applications doivent informer l'utilisateur de l'état final résultant d'un traitement. Le programme affiche alors à l'écran un message d'information.
L'application suivante montre l'utilisation de chaînes de caractères, déclarées sous la forme de constantes, pour stocker et gérer les messages. Cette application met en oeuvre un vecteur, des procédures, et différentes structures conditionnelles et répétitives. Elle synthétise assez bien l'état de nos connaissances à ce point de l'ouvrage.

APPLICATION V-3

Une Agence de Voyage affrète un avion de 24 places pour un voyage organisé. Analyser le problème en représentant l'appareil par un Vecteur de dimension 24, permettant la réservation et l'annulation des places, ainsi que la consultation du nombre et des numéros de places disponibles. Etablir l'algorithme et le programme. On utilisera des messages chaînes de caractères affichant à l'écran le résultat des opérations.

a) Analyse

– L' appareil est représenté par le Vecteur *vol*, de dimension *capa = 24*. Les éléments du Vecteur contiennent soit la valeur 0 (place libre), soit la valeur 1 (place réservée). Le choix d'une option (R pour Réserve, A pour Annule, C pour consulte, ou F pour Fin) appellera la procédure adéquate.
– La **réservation** d'une place nécessite la recherche du premier élément nul du vecteur. Si on en trouve un, sa valeur passe de 0 à 1, et on affiche le message *'Reservation effectuée'* suivi de la position de la place, indice de l'élément.
– L'**annulation** d'une réservation se pratique par la saisie du numéro de la place à libérer. L'élément se trouvant à l'indice égal au numéro de place passe de 1 à 0. Le message *'Annulation effectuee'* s'affiche suivi du numéro libéré.
– La **consultation** du nombre et des numéros de places disponibles s'effectue par balayage du Vecteur. On affiche, pour un élément nul, le message *'Non reservee'*, suivi de l'indice de cet élément Un compteur additionne les places libres et restitue le total en fin de balayage.

b) Algorithme

```
Début algorithme voyage
  entier :          /* place = indice des vecteurs */
    place, capa = 24, vol[1..capa]
  caractère :  option
  chaîne de caractères :
    messageA = 'Annulation effectuée  '
    messageR = 'Réservation effectuée '
    messageP = 'Place numéro  '
    messageN = 'Non réservée  '
    messageT = 'Total des places libres  '
Début action principale
  Pour place de 1 à capa faire
    vol[place] <--- 0          /* Initialisation */
  Fin Pour
  Répéter choix(option) Jusqu'à option = 'F'
Fin action principale
```



```
Action choix (VAR decide : caractère)
Début action
  écrire 'Taper R (Réserve),A (Annule) ou '
  écrire 'C (Consulte) ou F (Fin) ', lire decide
  Selon que decide
    'R' : reserve
    'A' : annule
    'C' : consulte
  Fin selon
Fin action

Action reserve
Début action
  place <--- 1
  Tant que place < capa ET vol[place] = 1 faire
    place <--- place +1
  Fin Tant que
  si vol[place] = 0
    alors  vol[place] <--- 1
           écrire messageR,messageP,place : 2
    sinon  écrire 'Vol Complet'
  Fin si
Fin action

Action annule
Début action
  Répéter
    écrire 'Saisir numéro place annulée'
    lire place
  Jusqu'à place > 0 ET place <= capa
  vol[place] <--- 0
  écrire messageA,messageP,place : 2
Fin action

Action consulte
  entier : j, compte
Début action
  compte <--- 0
  Pour j de 1 à capa faire
    si vol[ j ] = 0 alors
      compte <--- compte +1
      écrire messageN,messageP,j : 2
    Fin si
  Fin Pour
  écrire messageT,compte : 2
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM voyage(INPUT,OUTPUT);
CONST  capa = 24;
       messageA = 'Annulation effectuee ';
       messageR = 'Reservation effectuee ';
       messageP = 'Place numero ';
       messageN = 'Non reservee ';
       messageT = 'Total des places libres ';
  VAR  place : INTEGER;   (* indice vecteur *)
       vol : ARRAY[1..capa] OF INTEGER;
       option : CHAR;

PROCEDURE reserve;
BEGIN
  place := 1;
  WHILE (place < capa) AND (vol[place] = 1) DO
                        place := place +1;
  IF vol[place] = 0 THEN
    BEGIN
      vol[place] := 1;
      WRITELN(messageR,messageP,place :2)
    END
    ELSE WRITELN('Vol Complet')
END;

PROCEDURE annule;
BEGIN
  REPEAT
    WRITE('Saisir numero place annulee');
    READLN(place)
  UNTIL (place > 0) AND (place <= capa);
  vol[place] := 0;
  WRITELN(messageA,messageP,place : 2)
END;

PROCEDURE consulte;
VAR j, compte : INTEGER;
BEGIN
  compte := 0;
  FOR j := 1 TO capa DO IF vol[j] = 0 THEN
    BEGIN
      compte := compte +1;
      WRITELN(messageN,messageP,j : 2)
    END;
  WRITELN(messageT,compte : 2)
END;

PROCEDURE choix(VAR decide : CHAR);
BEGIN
  WRITE('Taper R (Reserve), A (Annule), ou ');
  WRITELN('C (Consulte) ou F (Fin)');
  READLN(decide);
  CASE decide OF
    'R' : reserve;
    'A' : annule;
    'C' : consulte;
  END
END;

BEGIN                    (* Programme principal*)
  FOR place := 1 TO capa DO
          vol[place] := 0;    (* Initialisation *)
  REPEAT
    choix(option)                (* 'R', 'A', 'C', 'F'*)
  UNTIL option = 'F'
END.
```

16. TABLEAUX A PLUSIEURS DIMENSIONS

Un tableau à N dimensions est une structure contenant des éléments de même type, dans laquelle chaque élément est repéré par N indices.

L'esprit humain peut facilement imaginer et manipuler des tableaux plans, à 2 dimensions ; il nous est également possible de penser avec des tableaux à 3 dimensions. Au delà, nous n'avons plus de schéma de représentation et cela devient un exercice difficile. Mais l'ordinateur n'a aucune difficulté pour gèrer des tableaux à 4, à 14 ou à N dimensions, la seule limite étant sa mémoire.

Restons concret. Nous limiterons cette étude aux tableaux à 2 dimensions, couramment utilisés, sachant que les tableaux de dimension supérieure sont traités de façon identique.

Un tableau à 2 dimensions correspond à la notion mathématique de **matrice**. Comme le montre la figure B-22, la 1ère dimension, horizontale, représente les lignes du tableau, et la seconde dimension, verticale, représente les colonnes.

Considérons une série d'objets dont une grandeur est mesurée, avec pour chaque objet, 2 repères entiers distincts.
- C'est le cas des notes de 30 élèves dans 8 matières (en précisant le numéro de l'élève entre 1 et 30, et le numéro de la matière entre 1 et 8)
- Ou plus généralement de valeurs de même type, contenues dans une matrice. Chaque valeur est identifiée par un numéro de ligne variant de 1 à lmax et un numéro de colonne variant de 1 à cmax.

Cette série de valeurs peut être représentée sous la forme mathématique d'une variable doublement indicée:

Nij pour des **notes**
  l'indice i correspond à l'élève (1 à 30)
  l'indice j correspond à la matière (1 à 8)
Vlc pour des **valeurs**
  l'indice l représente la ligne (1 à lmax)
  l'indice c représente la colonne (1 à cmax)

Ce type de variable est définie à l'aide d'un Tableau à 2 Dimensions, contenant lmax lignes et cmax colonnes. Chaque case du tableau, contenant une valeur, est repérée par 2 indices l et c, (l variant de 1 à lmax et c de 1 à cmax).

colonne / ligne	1	2	3	4	5
1	3	8	7	1	4
2	7	0	5	2	5
3	6	9	3	8	2

Fig. B-22. Tableau à 2 Dimensions.

Le tableau schématisé ci-dessus est constitué de 3 lignes et 5 colonnes et comporte 15 éléments du type entier.

Dans le cas des notes, l'identificateur du tableau peut être ***note*** (ATTENTION **not** est un mot réservé). On accède à la note de l'élève numéro 2 dans la matière 4 par la notation ***note[2,4]***, et plus généralement à la valeur de la ligne ***l*** dans la colonne ***c*** par la notation ***valeur[l,c]***.

Notation algorithmique de tableau

Comme pour la déclaration d'un vecteur, le type des éléments est donné suivi du symbole **:** , puis vient l'identificateur de la structure, et entre crochets, les plages des indices, séparées par des virgules.

Exemple : **réel : Note[1..30,1..8]**

déclare un tableau à 2 dimensions, comportant 30 lignes et 8 colonnes , et permettant le stockage en mémoire de 240 nombres réels. Chaque nombre est repéré par 2 indices entier, le premier compris entre 1 et 30 et le second entre 1 et 8.

Syntaxe Pascal

Elle est analogue à la syntaxe de déclaration d'un vecteur. Cependant 2 cas peuvent se présenter :

- Soit *type indice* désigne des indices multiples,
- Soit *type indice* est un indice unique (comme pour un vecteur) et dans ce cas, *type élément* est un type vecteur. Ce second cas correspond donc à la définition de Vecteur de Vecteur.

Exemple : Note : **ARRAY** [1..10,1..8] OF REAL;

peut s'écrire sous la forme

Note : **ARRAY**[1..10] OF **ARRAY**[1..8] OF REAL;

- De même, pour déclarer une variable correspondant à 25 lignes de texte de 80 caractères, on peut opter pour l'une ou l'autre des 2 formes suivantes :

Page : **ARRAY** [1..25,1..80] OF CHAR;
OU
Ligne : **ARRAY** [1..80] OF CHAR;
Page : **ARRAY** [1..25] OF Ligne;

- Imaginons que l'on souhaite gérer un Vecteur contenant 20 noms comportant chacun 15 caractères. Cet exercice mettra en oeuvre un Vecteur contenant des chaînes de 15 caractères qui sera déclaré de la manière ci-dessous :

Nom : **PACKED ARRAY** [1..15] OF CHAR;
Vecteurnom : **ARRAY** [1..20] OF Nom;

APPLICATION V-4

Une entreprise commerciale vend 5 produits à ses distributeurs, localisés sur 4 régions. Les prix de vente sont donnés ci-dessous, en fonction du produit et de la région.

N° de Région / N° de Produit	1	2	3	4
1	16	18	15	14
2	41	43	39	40
3	26	25	26	27
4	74	78	75	74
5	32	33	29	31

Etablir un algorithme et le programme réalisant :

- La saisie de ces valeurs dans un tableau à 2 dimensions,
- Le calcul du montant d'une commande passée par un distributeur situé dans la région R, pour X unités commandées du produit P.
- La génération de plusieurs calculs de commandes, avec un critère d'arrêt, au choix du programmeur. Faire une courte analyse préalable explicitant ce choix.

a) Analyse

Après avoir saisi les prix (Remplissage du tableau, soit par lecture, soit par affectation), le programme calcule le montant des commandes à partir des variables *region*, *produit* et *unites* commandées.
On réalisera une boucle sur la saisie de la région, contrôlée entre 0 et 4, sachant que la saisie 0 provoque la sortie de la boucle et l'arrêt du programme. On opèrera également un contrôle de saisie sur le Numéro du produit, entre 1 et 5.

b) Algorithme

```
Début algorithme Distributeur
  entier : region, produit, unites
  réel :   Prix[1..5,1..4], montant

Début action principale
  SaisiTableau
  SaisiRegion
  Tant que region <> 0 faire
    SaisiProduit
    SaisiUnites
    CalculMontant
    SaisiRegion
  Fin Tant que
Fin action principale

Action SaisiTableau
  entier : NoRegion, NoProduit
Début action
  Pour NoProduit de 1 à 5 faire
    écrire 'POUR LE PRODUIT', NoProduit
    Pour NoRegion de 1 à 4 faire
      écrire 'Saisir prix Région', NoRegion
      lire Prix[NoProduit, NoRegion]
    Fin Pour
  Fin Pour
  /* On pourrait éviter cette saisie, en */
  /* initialisant le tableau par affectation : */
  /* Prix[1,1] <--- 16, Prix[1,2] <--- 18, etc ...*/
  /* etc ... en terminant par  Prix[5,4] <--- 31 */
Fin action

Action SaisiRegion
Début action
  écrire                          /* saut de ligne */
  Répéter
    écrire 'Saisir Région (de 1 à 4)'
    écrire '(0 pour arrêter)
    lire region
  Jusqu'à region > -1 et region < 5
Fin action
```

```
Action SaisiProduit
Début action
  Répéter
    écrire 'Saisir Produit (de 1 à 5)'
    lire produit
  Jusqu'à produit > 0 et produit < 6
Fin action

Action SaisiUnites
Début action
  écrire 'Saisir Quantité'
  lire unites
Fin action

Action CalculMontant
Début action
  montant <--- unites * Prix[produit,region]
  écrire 'Montant calculé = ', montant :5:2
                                  /* saut de ligne */
  écrire
Fin action
```

c) Programme Pascal

```
PROGRAM Distributeur (INPUT,OUTPUT);
VAR   region, produit, unites : INTEGER;
      montant : REAL;
      Prix : ARRAY[1..5,1..4] OF REAL;

PROCEDURE SaisiTableau;
VAR
  NoRegion, NoProduit : INTEGER;
BEGIN
  FOR NoProduit := 1 TO 5 DO
  BEGIN
    WRITELN('POUR LE PRODUIT ',NoProduit);
    FOR NoRegion := 1 TO 4 DO
    BEGIN
      WRITELN('Saisir prix Region', NoRegion);
      READLN(Prix[NoProduit, NoRegion])
    END
  END
(* On pourrait eviter cette saisie, en *)
(* initialisant le tableau par affectation : *)
(* Prix[1,1] := 16, Prix[1,2] := 18, etc ...*)
(* etc ... en terminant par  Prix[5,4] := 31 *)
END;

PROCEDURE SaisiRegion;
BEGIN
  WRITELN;                    (* saut de ligne *)
  REPEAT
    WRITELN('Saisir Region (de 1 à 4)');
    WRITELN('(0 pour arreter)');
    READLN(region)
  UNTIL (region > -1) AND (region < 5)
END;

PROCEDURE SaisiProduit;
BEGIN
  REPEAT
    WRITELN( Saisir Produit (de 1 à 5)');
    READLN(produit)
  UNTIL (produit > 0) AND (produit < 6)
END;

PROCEDURE SaisiUnites;
BEGIN
  WRITELN('Saisir Quantite');
  READLN(unites)
END;

PROCEDURE CalculMontant;
BEGIN
  montant := unites * Prix[produit,region];
  WRITELN('Montant calcule = ', montant :5:2);
  WRITELN                     (* saut de ligne *)
END;

BEGIN                         (* Action Principale *)
  SaisiTableau;
  SaisiRegion;
  WHILE region <> 0 DO
    BEGIN
      SaisiProduit;
      SaisiUnites;
      CalculMontant;
      SaisiRegion
    END
END.
```

Remarque : Tant que le numéro de région saisi est différent de zéro, on réalise les actions *SaisiProduit, SaisiUnites, CalculMontant,* puis on relance *SaisiRegion. NoProduit* correspond aux lignes du Tableau *Prix* et *NoRegion* aux colonnes.

EXERCICE V-2 : Calcul du produit matriciel, dans un tableau TC[i,k], de 2 tableaux TA[i,j] et TB[j,k] de 4 lignes et 4 colonnes.
On rappelle TC[i,k] = Σ_j (TA[i,j] * TB[j,k]).

EXERCICE V-3 : Bataille Navale. 5 navires sont matérialisés dans un tableau carré de 6 lignes et 6 colonnes. Les navires N°1 et N°2 occupent 2 cases, les navires N°3 et N°4, 3 cases, et le navire N°5, 4 cases. Les cases représentant la mer contiennent 0, et celles correspondant à un navire contiennent le N° du navire. Le joueur dispose au maximum de 22 tirs. En fonction du coup joué, s'affiche un message :
'Tir déjà joué', 'Tir dans la mer',
'Navire N°',N,'Touché', 'Navire N°',N,'Coulé',
'GAGNE, tous coulés', 'PERDU, 22 tirs joués'.

17. ENSEMBLES

■ DEFINITION ET DECLARATION

Le Pascal définit le type ensemble comme une **collection finie d'objets de même nature.** Il est construit à partir des types simples (sauf réel), scalaire énuméré, ou intervalle.
Un ensemble se déclare par les mots **SET OF,** notés en Algorithmique par 'Ensemble de'.
SET OF peut apparaître au niveau des déclarations de TYPE, ou directement pour définir une variable dans la section VAR.

Syntaxe Pascal et Diagramme syntaxique

<type ensemble> ::= SET OF <type élément>

Exemples :

```
TYPE
    maxi9 = 1 .. 9;
    metal = (Fer, Chrome, Nickel, Etain);
    Outil = (tournevis, scie, cle, pince);
    reponse = SET OF 'N'..'O';
    alliage = SET OF metal;
    trousse = SET OF outil;

VAR
    chiffre, impair, pair : SET OF maxi9;
    Acier : alliage;
    materiel : trousse;
    decision : reponse;
```

Dans ces exemples, *alliage, trousse* et *reponse* sont des types ensemble, tandis que *Acier, materiel* et *decision* sont des variables correspondant à ces types.

Exemples de Notation algorithmique

```
Ensemble de 1 .. 9 : chiffre,impair,pair
Ensemble de (Fer,Chrome,Nickel,Etain) : Acier
```

Une variable de type ensemble peut prendre pour valeur une, plusieurs ou toutes les valeurs de son type.

Les valeurs d'une variable ensemble sont énumérées entre crochets [ ], et séparées par des virgules. On peut aussi utiliser la représentation intervalle .. lorsque l'ensemble contient plusieurs valeurs consécutives d'un type scalaire.
Dans le corps du programme, on pourra réaliser les affectations suivantes :

```
Chiffre := [1..9];
impair := [5,3];
Acier := [Fer .. Nickel];
(* équivalent à [Fer, Chrome, Nickel] *)
materiel:= [cle, tournevis, pince];
pair := [ ];      (* Valeur ensemble vide *).
decision := ['N'];
```

■ OPERATEURS SUR LES ENSEMBLES

Les éléments d'un ensemble ne sont pas accessibles individuellement. Leurs assemblages résultent des opérations classiques de la Théorie Mathématique (Intersection, Union, Différence).
Le Tableau ci-dessous montre que les autres opérateurs ensemblistes produisent des résultats logiques (Booléen).

Opérateur	Symbole	Type du Résultat
Affectation	:=	Ensemble
Intersection	*	Ensemble
Union	+	Ensemble
Différence	-	Ensemble
Egalité	=	Logique
Inégalité	<>	Logique
Inclusion	<= >=	Logique
Appartenance	IN (DANS)	Logique

L'opérateur d'appartenance permet de tester si une valeur donnée appartient à un ensemble. Il peut être utilisé dans les contrôles de saisie, comme le montre l'application suivante.

■ EXEMPLE NUMERIQUE SIMPLE

Chiffre, impair, pair, serie, suite étant des variables de type *maxi9 = 1 .. 9,*

1.- Calculer l'ensemble des nombres pairs par différence entre l'ensemble des chiffres [1..9] et de l'ensemble des nombres impairs [1,3,5,7,9].

2.- **Calculer l'intersection** des variables *serie* (valeur [1..5]) et *suite* (valeur [4,5,6,7,8,9]).
Afficher les résultats.

3.- **Calculer l'union** des variables *serie* et *impair.* Afficher les résultats.

a) Algorithme

```
Début algorithme ensemble
  ensemble de 1..9 : chiffre,impair,pair,
                     serie,suite
  entier : i
Début action
  chiffre <--- [1..9], impair <--- [1,3,5,7,9]
  pair <--- chiffre - impair
  serie <--- [1..5], suite <--- [4,5,6,7,8,9]
  chiffre <--- serie * suite
  Pour i de 1 a 9 faire si i DANS chiffre alors
    écrire i :2, ' dans ensemble 4,5'
  Fin Pour
  chiffre <--- serie + impair
  Pour i de 1 a 9 faire si i DANS chiffre alors
    écrire i :2, ' dans ensemble 1,2,3,4,5,7,9'
  Fin Pour
Fin action
Fin algorithme
```

b) Programme Pascal

```
PROGRAM ensemble(INPUT,OUTPUT);
TYPE  maxi9 = 1..9;
VAR
  chiffre,impair, pair, serie, suite : SET OF maxi9;
  i : INTEGER;
BEGIN
  chiffre := [1..9];  impair := [1,3,5,7,9];
  pair := chiffre - impair;
  serie := [1..5];  suite := [4,5,6,7,8,9];
  chiffre := serie *  suite;
  FOR i := 1 TO 9 DO IF i IN chiffre THEN
    WRITELN(i:2,' dans ensemble 4,5');
  chiffre := serie + impair;
  FOR i := 1 TO 9 DO IF i IN chiffre THEN
    WRITELN(i:2,' dans ensemble 1,2,3,4,5,7,9')
END.
```

```
(* INSTRUCTION *)          (* RESULTAT *)

Chiffre := [1..9];         (* Tous de 1 à 9 *)
impair :=  [1,3,5,7,9];    (* 1,3,5,7,9 *)
pair := Chiffre - impair;  (* 2,4,6,8 *)
serie := [1..5];           (* 1,2,3,4,5 *)
suite := [4,5,6,7,8,9];    (* 4,5,6,7,8,9 *)
Chiffre := serie * suite;  (* 4,5 *)
Chiffre := serie + impair; (* 1,2,3,4,5,7,9 *)

serie = suite              (* FAUX *)
serie <> suite             (* VRAI *)
serie <= Chiffre           (* VRAI *)
nombre := 5;
nombre IN pair             (* FAUX *)
```

APPLICATION V-5

Un dispositif d'alimentation automatique remplit une bétonnière pour préparer un mortier, mélange de ciment (ou de chaux), de sable, de gravier et d'eau.
L'utilisateur saisit au clavier une lettre correspondant au nom de l'élément à charger (contrôlé dans l'ensemble des 5 éléments possibles) puis saisit le poids pour cet élément (50 à 200 kg).
Sachant que le poids total du mélange ne peut excéder 500 kg, établir un algorithme et le programme permettant d'effectuer ces saisies.

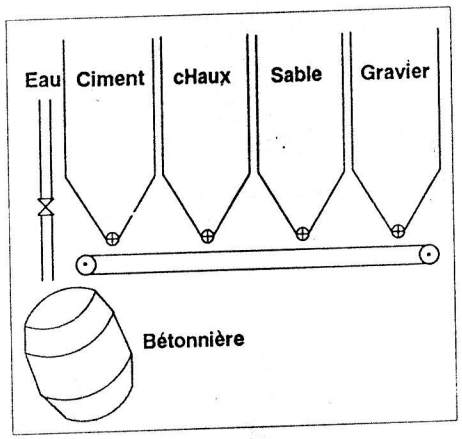


Fig. B-23. Préparation d'un Mortier

a) Algorithme

```
Début algorithme Mortier
  entier :  NbCompos,   /* Nombre Composants*/
            poids,poitotal
  ensemble de caractères : element
  ensemble de 50..200 : plage

Début action principale
  Initialise
  SaisiNombre
  SaisiElement
Fin action principale

Action Initialise
Début action
  element <--- ['C', 'H', 'S', 'G', 'E']
  plage <--- [50 .. 200]
  poitotal <--- 0
Fin action
```



```
Action SaisiNombre
Début action
  Répéter
    écrire 'Nombre de Composants (3 a 4)'
    lire NbCompos
  Jusqu'à NbCompos > 2 ET NbCompos < 6
Fin action

Action SaisiElement
  entier : i
  caractère : Majuscule
Début action
  poitotal <--- 0
  Pour i de 1 à NbCompos faire
    Répéter
      écrire 'Ciment, cHaux, Sable, Gravier,Eau'
      écrire i :1, ' Saisir Majuscule Composant', i
      lire Majuscule
    Jusqu'à Majuscule DANS melange
    écrire                              /* saut de ligne */
    SaisiPoids(i)
  Fin Pour
Fin action

Action SaisiPoids(numero : entier)
Début action
  Répéter
    écrire 'Poids Elément ', numero,' (50..200)'
    écrire 'Reste Maxi = ', 500 - poitotal
    lire poids
  Jusqu'à poids DANS plage et poitotal <= 500
  poitotal <--- poitotal + poids
  écrire                            /* saut de ligne */
Fin action

Fin algorithme
```

Remarque : L'action procédure *SaisiPoids* reçoit le paramètre par valeur correspondant au numéro de composant traité. *SaisiPoids* informe l'utilisateur de la quantité restant disponible dans la bétonnière.

b) Programme Pascal

```
PROGRAM Mortier (INPUT,OUTPUT);
TYPE
  melange = SET OF CHAR;
  masse = SET OF 50..200;
VAR
  NbCompos,poids,poitotal : INTEGER;
  element : melange;
  plage : masse;

PROCEDURE SaisiPoids(numero : INTEGER);
BEGIN
  REPEAT
  WRITELN('Poids Element ',numero,' (50..200)');
    WRITELN('Reste Maxi = ',500 - poitotal);
    READLN(poids);
  UNTIL (poids IN plage) AND
            ((poitotal + poids) <= 500);
  poitotal := poitotal + poids;
  WRITELN
END;

PROCEDURE SaisiElement;
VAR   i : INTEGER;
      Majuscule : char;
BEGIN
  poitotal := 0;
  FOR i := 1 TO NbCompos DO
    BEGIN
      REPEAT
  WRITELN('Ciment, cHaux, Sable,Gravier,Eau');
  WRITELN(i :1,' Saisir Majuscule Composant');
        READLN(Majuscule)
      UNTIL Majuscule IN element;
      WRITELN;                    (* saut de ligne *)
      SaisiPoids(i)
    END
END;

PROCEDURE SaisiNombre;
BEGIN
  REPEAT
    WRITELN('Nombre de Composants (3a4)');
    READLN(NbCompos)
  UNTIL (NbCompos > 2) AND (Nbcompos < 6);
WRITELN                           (* saut de ligne *)
END;

PROCEDURE Initialise;
BEGIN
  element := ['C', 'H', 'S', 'G', 'E'];
  plage := [50 .. 200];  poitotal := 0
END;

BEGIN                             (* action principale *)
  Initialise;
  SaisiNombre;
  SaisiElement
END.
```

EXERCICE V-4 : Construire 2 ensembles VOY et CONS, l'un contenant les voyelles et l'autre les consonnes 'L', 'M', 'N' 'R', 'S', 'T'. Réaliser l'Union de ces 2 ensembles et demander la saisie d'une chaîne de caractères en n'autorisant que les caractères appartenant à VOY Union CONS.

18. STRUCTURE D'ENREGISTREMENT

■ DEFINITION ET DECLARATION

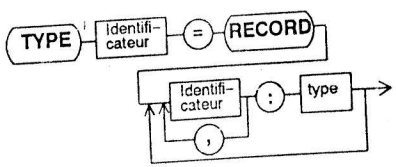
Les structures de données étudiées jusqu'ici, ont permis l'utilisation des types Vecteur, Tableau et Ensemble. Ces structures ont toutes un point commun : Les variables engendrées à partir de ces structures contiennent un nombre fini d'**éléments de même type**.

Considérons maintenant l'objet **produit**, identifié par une **référence (valeur entière)**, et comportant les informations suivantes : le nom du produit, ou **libellé (chaîne de caractères)**, et son prix de vente Hors Taxe, **prixht (valeur réelle)**.
Cette structure mélange une valeur de type entier, une valeur de type chaîne de caractères et une valeur de type réel. On ne peut donc pas représenter cet objet sous la forme d'un vecteur.
Pour le définir, nous introduisons un nouveau type, le **TYPE ENREGISTREMENT** également nommé **TYPE ARTICLE**.

Le Pascal déclare le type enregistrement par le mot **RECORD** suivi du signe = ; viennent ensuite les identificateurs des différents composants ou CHAMPS de la structure, avec leur type. Le type enregistrement se termine par le mot **END;**
Le mot RECORD peut apparaître au niveau des déclarations de TYPE, ou directement pour définir une variable dans la section VAR.

Syntaxe Pascal et Diagramme syntaxique

```
<type enregistrement> ::=
   RECORD
      <identif.du champ1> : <type du champ1>
      ...
      <identif.du champn> : <type du champn>
   END;
```



Exemples :

```
TYPE
  produit =
    RECORD
      reference : INTEGER;
      libelle : packed ARRAY [1..15] of CHAR;
      prixht : REAL
    END;

VAR
    article :   produit;
    cuve :      RECORD
                  numero : INTEGER;
                  volume : REAL;
                  vide : BOOLEAN
                END;
```

Dans ces exemples, *produit* est un type enregistrement, tandis que *article* est une variable correspondant à ce type. Notons également que la définition du type de la variable *cuve* est réalisée par la déclaration de la variable ; *cuve* contient les champs *numero*, *volume* et *vide*.
Une variable de type enregistrement contient plusieurs CHAMPS, de différents types .

Exemple de Notation algorithmique

```
Enregistrement : cuve
   entier :    numero
   réel :      volume
   logique :   vide
Fin Enregistrement
```

■ LECTURE ET ECRITURE DES CHAMPS

Le Programmeur accède aux champs d'une variable enregistrement en précisant le nom de la variable, suivi d'un point, et du nom du champ.

variable enregistrement . nom du champ

Dans le corps du programme, on pourra réaliser les affectation, écritures et lectures suivantes :

Exemples :

```
article . reference := 101;
WRITELN('reference = ',article . reference);
WRITELN('Saisir Prix Hors Taxe');
READLN(article . prixht);
WRITELN('Saisir libelle');
saisichaine(15, article . libelle);
```


La procédure *saisichaine* , permettant la lecture de la chaîne *article.libelle*, a été définie à propos des vecteurs chaînes de caractères (page 73). Cette procédure possède deux paramètres, l'un mentionne la longueur de la chaîne, l'autre retourne le contenu de la chaîne saisie.

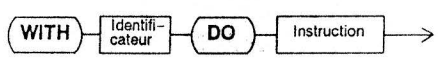
■ L'INSTRUCTION WITH

Une seconde méthode, plus simple, permet d'accéder aux champs d'une variable enregistrement. Cette méthode consiste à stipuler AVEC quelle variable enregistrement on souhaite travailler, pour ne mentionner, dans la structure WITH, que les identificateurs de champs.

Dans l'instruction WITH , les champs sont désignés par leurs noms sans être précédés du nom de la variable enregistrement.

Syntaxe et Diagramme du WITH

```
WITH <identif. d'enregistrement> DO
  BEGIN
    <identif.du champ1> := <valeur1>;
    ...
    <identif.du champn> := <valeurn>
  END
```



Exemples Algorithmique et Pascal

```
Avec cuve faire
   numero <--- 11
   écrire 'Saisir volume'
   lire(volume)
Fin Avec
```

```
WITH cuve DO
  BEGIN
    numero := 11;
    WRITELN('Saisir volume');
    READLN(volume)
  END;
```

AVEC la variable *cuve*, on intervient sur ses champs *numero*, *volume*, *vide*, pour leur donner (par affectation, ou par lecture) une des valeurs correspondant à leur type.

■ IMBRICATION

Soit une variable RECORD, nommée *rec1*, possédant un champ de type RECORD, nommé *rec2*, ce dernier composé des champs *ch21* et *ch22* de type entier, on pourra affecter à ces champs les valeurs 5 et 6 par les instructions:

```
(* 1 *)     rec1 . rec2 . ch21 := 5;
(* 2 *)     WITH rec1 DO
                WITH rec2 DO  ch22 := 6;
```

Les deux formes sont correctes ; la première se fait par niveau, tandis que la seconde montre l'imbrication possible des instructions WITH.

■ ENREGISTREMENT AVEC VARIANTE

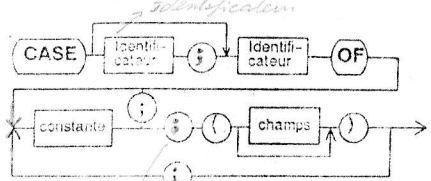
Une variable de type enregistrement peut être composée de 2 parties, **une partie fixe**, que nous venons d'étudier, **et une partie variante**, pour laquelle un champ joue le rôle de sélecteur.

Le sélecteur, de type entier, caractère, scalaire énuméré ou intervalle, utilise la syntaxe du CASE (Cf page 35). La structure de la partie variante est déterminée en fonction des différentes valeurs prises par le sélecteur.

Syntaxe et Diagramme de la partie VARIANTE

```
<type enregistrement> ::=
RECORD
  <champ1> : <typechamp1>;
  CASE <champ2> : <typechamp2> OF
     valeur1 : ( <champ21> :<typechamp21>;
                 <champ2n> :<typechamp2n> ) ;
     valeurx : ( <champ2x> :<typechamp2x> )
  END
```

Dans tous les cas d'enregistrement avec variante, le **END** terminal de la structure enregistrement est aussi terminal da la partie CASE. Autrement dit, la partie fixe doit précéder la partie variante.



La partie fixe peut contenir plusieurs champs.
La partie variante, qui par définition contient plusieurs champs, doit être unique et terminale, mais elle peut imbriquer d'autres parties variantes.

Exemple :

Considérons le type enregistrement *matiere* :
– **La partie fixe** est composée du *nom* (chaîne de 15 caractères), de la ***densité*** **(réel)** et de l'*aspect* (scalaire énuméré : solide, liquide ou gazeux)
– **La partie variable** est fonction de ***aspect***.
S'il est **solide**, on note le *point de fusion*,
S'il est **liquide**, *le point d'ébullition* et *le point de congélation*
S'il est **gazeux**, *le volume* et *la température*.

Les déclarations nécessaires à la définition du type *matière* sont données ci-dessous.

```
TYPE
    etat = (solide, liquide, gazeux);
    matiere =
      RECORD
        nom : packed ARRAY[1..15] of char;
        densite : REAL;
        CASE aspect : etat OF
          solide : (fusion : REAL);
          liquide: (ebull, congel : REAL);
          gazeux : (volume : REAL;
                    temper : INTEGER)
      END;
```

Lorsque la valeur du sélecteur correspond à une liste de champs vide il faut écrire : **( )**.

Pour la variable *resultat* de type *matiere*, seront saisis les champs *nom* , *densite*, et *aspect*, qui tous trois appartiennent à la partie fixe. Le champ *aspect* de type *etat*, ne peut prendre qu'une des 3 valeurs énumérées. On saisira ensuite le ou les sous-champs correspondant à la valeur de *aspect*. Si *aspect = gazeux*, on saisira un nombre réel pour le sous-champ *volume* et un entier pour le sous-champ *temper*.

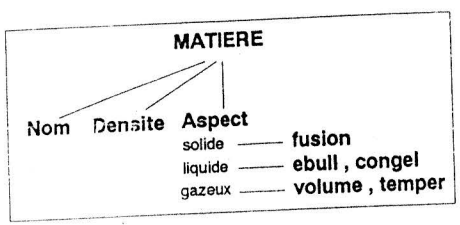


Fig. B-24. Schéma d'enregistrement matière

■ OPERATIONS SUR UN ENREGISTREMENT

Les différents champs d'une variable enregistrement sont manipulés isolément, l'un après l'autre. Le seul cas où la variable peut être manipulée **dans son ensemble** concerne l'instruction d'affectation . Elle est possible à partir d'une autre variable de même type.

```
V2 := V1;
```

Si V1 et V2 sont 2 variables du même type enregistrement, on pourra écrire que V2 prend la valeur de V1. Les champs et sous-champs de V2 prendront alors pour valeurs celles des champs et sous-champs de V1.

APPLICATION V-6

Saisir les constituants d'une tâche, variable enregistrement composée des champs :
– numéro (de type entier)
– début (de type temps)
– durée (de type temps).
Le type temps est à définir comme un enregistrement contenant les champs :
– heure (de 0 à 23)
– minute (de 0 à 59)
– seconde (de 0 à 59)
Calculer et afficher en heure, minute, seconde, l'instant correspondant à la fin de cette tâche.

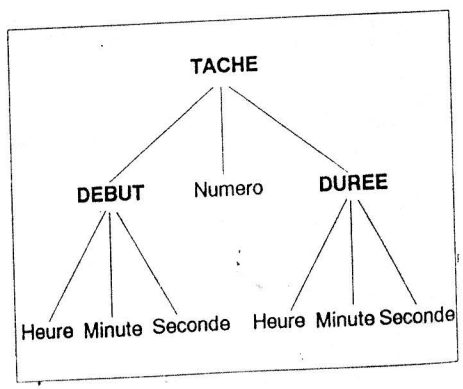


Fig. B-25. Schéma d'enregistrement tâche

a) Algorithme

```
Début algorithme travail
  entier : H, M, S      /* heure, minute, seconde */
  Enregistrement : tache
      entier : numero
      Enregistrement : debut
          intervalle 0 .. 23 : heur
          intervalle 0 .. 59 : minut
          intervalle 0 .. 23 : sec
      Fin enregistrement
      Enregistrement : duree
          intervalle 0 .. 23 : heur
          intervalle 0 .. 59 : minut
          intervalle 0 .. 59 : sec
    Fin enregistrement
  Fin enregistrement

Début action
  Avec tache faire
    écrire 'Saisir numéro de la tache'
    lire numero
    écrire 'DEBUT de la tache'
    Avec debut faire
      écrire 'Saisir heure (0..23)'
      lire heur
      écrire 'Saisir minute (0..59)'
      lire minut
      écrire 'Saisir seconde (0..59)'
      lire sec
    Fin Avec              /* de début de tache */
    écrire 'DUREE de la tache'
    Avec duree faire
      écrire 'Saisir heure (0..23)'
      lire heur
      écrire 'Saisir minute (0..59)'
      lire minut
      écrire 'Saisir seconde (0..59)'
      lire sec
    Fin Avec              /* de durée de tache */
  Fin Avec                /* de tache */
  H <--- tache.debut.heur + tache.duree.heur
  M <--- tache.debut.minut + tache.duree.minut
  S <--- tache.debut.sec + tache.duree.sec
  écrire 'Instant de FIN de tache : '
  écrire H :2, ' heure(s) '
  écrire M :2, ' minute(s) '
  écrire S :2, ' seconde(s)'
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM travail(INPUT,OUTPUT);
TYPE    temps = RECORD
                  heur   : 0 .. 23;
                  minut  : 0 .. 59;
                  sec    : 0 .. 59
                END;
VAR     H, M, S : INTEGER;
        (* heure, minute, seconde *)
        tache : RECORD
                  numero : INTEGER;
                  debut : temps;
                  duree : temps
                END;
BEGIN
  WITH tache DO
    BEGIN
      WRITELN('Saisir numero de la tache');
      READLN(numero);
      WRITELN('DEBUT de la tache');
      WITH debut DO
        BEGIN
          WRITE('Saisir heure (0..23) ');
          READLN(heur);
          WRITE('Saisir minute (0..59) ');
          READLN(minut);
          WRITE('Saisir seconde (0..59) ');
          READLN(sec)
        END;              (* de debut de tache *)
      WRITELN('DUREE de la tache');
      WITH duree DO
        BEGIN
          WRITE('Saisir heure (0..23) ');
          READLN(heur);
          WRITE('Saisir minute (0..59) ');
          READLN(minut);
          WRITE('Saisir seconde (0..59) ');
          READLN(sec)
        END               (* de duree de tache *)
    END;                  (* de tache *)
  H := tache.debut.heur + tache.duree.heur;
  M := tache.debut.minut + tache.duree.minut;
  S := tache.debut.sec + tache.duree.sec;
  WRITE('Instant de FIN de tache : ');
  WRITE(H :2, ' heure(s) ');
  WRITE(M :2, ' minute(s) ');
  WRITE(S :2, ' seconde(s) ')
END.
```

EXERCICE V-5 : Définir un enregistrement Date de la forme JJMMAA (0 ..31, 0 .. 12, 0 ..99) et l'intégrer dans la définition et la saisie d'une variable enregistrement Elève définie par les nom, prénom, date de naissance, N° de classe et série.

19. POINTEURS

■ DEFINITION ET DECLARATION

Toutes les variables étudiées dans les chapitres précédents étaient, pour un programme donné, connues en nombre, nommées et déclarées en début de bloc.

La possibilité de **créer de nouvelles variables, pendant l'exécution du programme**, est donnée en Pascal par l'introduction de **variables dynamiques**, gérées en fonction des besoins.
Ces variables dynamiques font appel à un nouveau type, le TYPE POINTEUR.

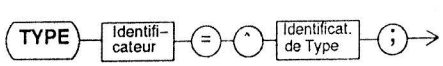
La mémoire centrale de l'ordinateur est constituée de 'cellules'. Chaque cellule est repérée par une adresse et possède un contenu. A chaque adresse est donc associée l'information contenue dans la cellule mémoire correspondante.

Le type POINTEUR représente un ensemble de valeurs, qui sont des adresses d'éléments d'un certain type.

La partie déclaration de type d'un programme pourra définir des types pointeurs sur les types entier, réel, caractère, logique, mais aussi sur des tableaux, et des enregistrements.
Un type pointeur se déclare par le symbole ^ , suivi du type des éléments à référencer.

Syntaxe et Diagramme du type POINTEUR

<type pointeur> ::= ^ <type éléments pointés>



Exemples :

```
TYPE
  ptentier  = ^ INTEGER;
  ptcarac   = ^ CHAR;

  lot       = RECORD
                numero  : INTEGER;
                poids   : REAL
              END;
  ptlot = ^ lot;

VAR
     pointen : ptentier;
     pointre : ^ REAL;
     pointca : ptcarac;
     pointbo : ^ BOOLEAN;
     pointlo : ptlot;
```

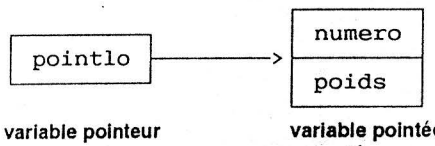
Dans ces exemples, nous avons volontairement déclarés les types pointeurs avec des identificateurs commençant par les lettres *pt*, et les variables pointeurs avec des identificateurs commençant par les lettres *point*.
Notons que les déclarations des variables *pointre* et *pointbo* définissent directement leur type.

Exemple de Notation algorithmique

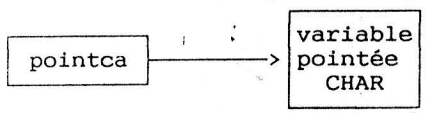
Pointeur d'entier : ptentier
Enregistrement : **lot** entier : numero réel : poids Fin Enregistrement
Pointeur de lot : ptlot

■ POINTEURS ET VARIABLES POINTEES

Une variable pointeur est plus simplement nommée pointeur.
Le pointeur *pointlo* est une variable pointeur contenant une adresse de structure de type *lot*.
On dit que *pointlo* pointe sur une **variable pointée** associée dont l'adresse est contenue dans *pointlo*.



variable pointeur — **variable pointée associée**



Le pointeur *pointca* contient l'adresse de la variable caractère pointé.

Un pointeur ou variable pointeur contient l'adresse d'une variable pointée associée.

Le pointeur est déclaré dans le programme, mais une (ou plusieurs) variable pointée sera créée dynamiquement dans le programme par la procédure standard NEW, que nous définissons ci-après.

1- LES PROCEDURES NEW ET DISPOSE

La procédure NEW (NOUVEAU) crée une nouvelle variable pointée. La procédure possède un paramètre pointeur, que nous appelons *varpointeur*, dans l'instruction ci-dessous.

NOUVEAU (varpointeur)
NEW (varpointeur);

NEW(pointca) réserve en mémoire la place nécessaire à un caractère pointé par *pointca*. L'adresse de ce caractère est affecté à *pointca*.
NEW(pointlo) réserve en mémoire la place pour une structure de type *lot*, et affecte l'adresse de cette structure à la variable pointeur *pointlo*.

La variable pointée, allouée dynamiquement en mémoire, n'a pas été déclarée et ne possède pas d'identificateur propre. Elle est repérée par la **notation pointée**, définie ci-dessous.

La variable pointée est notée *varpointeur* ^

pointca ^ identifie la variable caractère pointée par le pointeur *pointca*, et la notation **pointlo ^** identifie la variable pointée par le pointeur *pointlo*.

La procédure prédéfinie **DISPOSE** (LIBERE) permet de libérer l'espace mémoire allouée à une variable pointée. DISPOSE possède un paramètre pointeur, soit *varpointeur*, déjà utilisé pour la création de la variable pointée.

LIBERE (varpointeur)
DISPOSE (varpointeur);

Nous savons maintenant créer, identifier et détruire les variables pointées. On peut donc effectuer sur ces variables, les opérations classiques (affectation, lecture, écriture) et les opérations spécifiques relatives à leur type (calculs sur les valeurs numériques).

2- EXEMPLE NUMERIQUE SIMPLE

Calcul de la valeur d'un lot, composé d'un *numero* de lot et d'un *poids*.
– Saisir le prix au kilo du lot concerné,
– Créer une variable pointée de type *lot* (*numero* : entier, *poids* : réel) et utiliser l'affectation, la lecture et l'écriture sur les champs de la variable pointée *pointlo* ^,
– Calculer et afficher la valeur du lot,
– Libérer la place mémoire allouée à *pointlo* ^.

a) Algorithme

```
Début algorithme pointeur1
  réel :prixkilo, valeur
  Enregistrement : lot
    entier : numero
    réel :  poids
  Fin enregistrement
  pointeur de lot : pointlo
Début action
  écrire 'Saisir le prix du kilo'
  lire prixkilo
  NOUVEAU(pointlo);
  pointlo ^ .numero <--- 4;
  écrire 'Saisir poids lot ',pointlo ^ .numero
  lire pointlo ^ .poids
  valeur <--- prixkilo * pointlo ^ .poids;
  écrire 'La valeur du lot est :', valeur :5:2
  LIBERE(pointlo)
Fin action
Fin Algorithme
```

b) Programme Pascal

```
PROGRAM pointeur1 (INPUT,OUTPUT);
TYPE
    lot   = RECORD
              numero   : INTEGER;
              poids    : REAL
            END;
VAR
     pointlo : ^ lot ;
     prixkilo, valeur : REAL;
BEGIN
  WRITELN('Saisir le prix du kilo');
  READLN(prixkilo);
  NEW(pointlo);
  pointlo ^ .numero := 4;
  WRITELN('Saisir poids lot ',pointlo ^ .numero);
  READLN(pointlo ^ .poids);
  valeur := prixkilo * pointlo ^ .poids;
  WRITELN('La valeur du lot est :', valeur :5:2);
  DISPOSE(pointlo)
END.
```

Remarque : On aurait pu utiliser la notation WITH

```
WITH pointilo ^ DO
  BEGIN
    numero : = 4;
    WRITELN('Saisir poids lot',numero);
    READLN(poids);
  END;
```

■ LES LISTES CHAINEES

On peut affecter à un pointeur soit la valeur d'un pointeur de même type, soit la **constante prédéfinie NIL.**
Le fait qu'un pointeur prenne la valeur NIL signifie qu'il ne pointe sur aucune variable pointée. C'est le cas du dernier élément d'une liste chaînée.

Plusieurs instructions successives NEW, portant **sur une même variable pointeur,** créent autant de variables pointées (zones mémoire réservées) qu'il y a d'instructions. Au fur et à mesure de ces créations, la variable pointeur prend pour valeur l'adresse de la dernière variable pointée créée, "écrasant" ainsi l'adresse précédente.

Si le programmeur souhaite conserver les adresses des variables pointées précédemment créées, il doit utiliser une particularité de la structure enregistrement, autorisant un chaînage.

Considérons une liste de 4 variables pointées, associées à la même variable pointeur, telles que leur structure contienne chacune 2 informations :
– Un premier champ contenant une information sur la valeur de la variable (ex : un caractère),
– Un second champ contenant une information sur l'adresse de la variable suivante dans la liste.
Le type pointeur sur la structure d'enregistrement suivante correspond à ces définitions :

```
TYPE
  ptcarchaine = ^ carchaine
  carchaine = RECORD
                car :      CHAR;
                suivant :  ptcarchaine
              END;
```

C'est le seul cas en Pascal où on utilise un identificateur (carchaine dans l'exemple), **pour ne le définir qu'après.**
Les variables pointées vont contenir chacune une information (caractère dans notre exemple) **et l'adresse de la variable pointée suivante. Ceci réalise un chaînage.**

APPLICATION V-7

Construire un algorithme et un programme réalisant une liste chaînée de 4 caractères, en respectant la méthode suivante :
1– Réserver par la procédure NEW (NOUVEAU) la place mémoire pour la 1ère variable pointée,
2– Lire le caractère correspondant à *point1* ,
3– Sauvegarder la valeur du pointeur correspondant dans la variable pointeur *debut*,
4– Puis exécuter 3 fois la séquence suivante :
41– Sauvegarder la valeur de la variable pointeur *point1* dans la variable pointeur *point2*,
42– Réserver par la procédure NEW une nouvelle place mémoire pour la prochaine variable pointée, puis lire le caractère,
43– Donner la nouvelle valeur de la variable pointeur au champ *suivant* (de type ptcarchaine) de *point2*,
5– Enfin donner au champ *suivant* de la dernière variable pointée la valeur **NIL**, puisqu'il ne pointera sur aucune autre variable.
6– Ecrire un boucle permettant d'afficher à l'écran les caractères saisis.

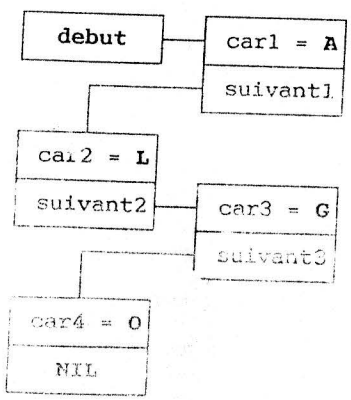
a) Algorithme

```
Début algorithme listechainee
  entier : i
  pointeur de carchaine : debut, point1, point2
  Enregistrement : carchaine
    caractère : car
    pointeur de carchaine : suivant
  Fin enregistrement
Début action
  NOUVEAU(point1)
  écrire 'Saisir premier caractère'
  lire point1 ^.car
  debut <--- point1
  Pour i de 2 à 4 faire
      point2 <--- point1
      NOUVEAU(point1)
      écrire 'Saisir caractère numéro', i
      lire point1 ^ . car
      point2 ^. suivant <--- point1
  Fin Pour
  point1 ^. suivant <--- NIL
  /* écriture de la liste chaînée des caractères */
  écrire                          /* saut de ligne */
  point1 <--- debut
  Tant que point1 <> NIL faire
    écrire point1 ^. car
    point1 <--- point1 ^. suivant
  Fin Tant que
Fin action
Fin Algorithme
```


b) Programme Pascal

```
PROGRAM listechainee(INPUT,OUTPUT);
TYPE
  ptcarchaine = ^ carchaine;
  carchaine = RECORD
                car :      CHAR;
                suivant :  ptcarchaine
              END;
VAR
  debut, point1, point2 : ptcarchaine;
  i : INTEGER;
BEGIN
  NEW(point1);
  WRITELN('Saisir premier caractere');
  READLN(point1 ^ . car);
  debut := point1;
  FOR i := 2 TO 4 DO
    BEGIN
      point2 := point1;
      NEW(point1);
      WRITELN('Saisir caractere numero', i);
      READLN(point1 ^ . car);
      point2 ^. suivant := point1
    END;
  point1 ^. suivant := NIL;
  (* ecriture de la liste chainee des caracteres *)
  WRITELN;                          (* saut de ligne *)
  point1 := debut;
  WHILE point1 <> NIL DO
    BEGIN
      WRITE(point1 ^. car);
      point1 := point1 ^. suivant
    END
END.
```

SCHEMA DE LA LISTE CHAINEE "ALGO"



■ AUTRES APPLICATIONS DES POINTEURS

Une liste chaînée d'éléments est une **représentation linéaire** dont chaque élément contient des données et un pointeur vers l'élément suivant.
Les pointeurs permettent également de modéliser des **structures non linéaires telles que les arbres.**
Dans une structure d'arbre, chaque élément, appelé noeud, contient des données et **deux** pointeurs (**arbre binaire**) ou **plusieurs** pointeurs (arbres complexes), vers les éléments suivants.

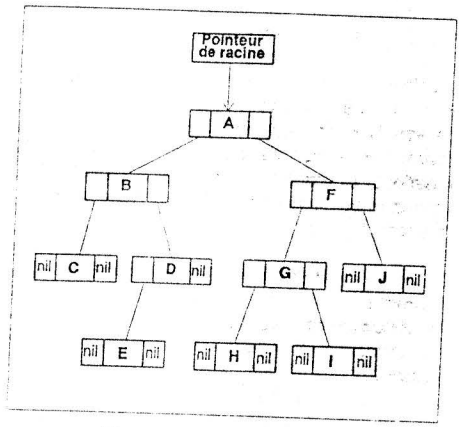


Fig. B–26. Arbre binaire

Dans l'arbre binaire de la figure B–26, la lecture est faite à partir de la racine, en accordant une priorité absolue à la descente vers la gauche.

Les pointeurs sont utilisés pour la résolution de problèmes dans les domaines variés de :
– L'analyse syntaxique, notamment mise en oeuvre dans les compilateurs qui vérifient le respect des règles grammaticales d'un langage.
– L'intelligence artificielle où interviennent de nombreux essais successifs (jeu d'échecs) dont l'objectif est d'obtenir un chemin optimisé.
– La Recherche Opérationnelle, incluant les Relations binaires, la Théorie des Graphes , les problèmes de Transport et d'Ordonnancement.

EXERCICE V–6 : Rechercher un élément de valeur VAL dans une liste chaînée, le supprimer en remettant à jour les pointeurs de la liste.

20. FICHIERS

DEFINITION ET DECLARATION

La notion de fichier est passée dans le grand public, comme concernant un ensemble d'informations sur des personnes ou des objets :

- Fichier des **clients** d'une entreprise,
- Catalogue de **produits**,
- Fichier des **cartes grises de véhicules**,
- Fichier de **pièces détachées** ...

Un fichier est un ensemble ordonné de composants présentant une même structure et pouvant être consultés individuellement.

La structure d'enregistrement peut, comme nous l'avons vue au chapitre 18, être utilisée indépendamment de la notion de fichier. Mais il suffit de se reporter à ce chapitre, pour imaginer la représentation, avec une variable RECORD, de

- **L'enregistrement ou fiche d'un client**, comportant les Nom, Adresse, Téléphone ...
- **L'enregistrement ou fiche d'un produit**, incluant le Numéro de référence, le libellé et le prix.
- **La fiche carte grise d'un véhicule**, avec mention de la Marque, du Type, de la date de Mise en circulation, du N° Immatriculation, du Nom et de l'adresse du propriétaire ...
- **L'enregistrement ou fiche d'une pièce**, contenant le code de l'atelier, la référence et le nom de la pièce, la quantité en stock ...

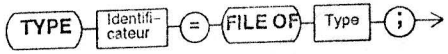
L'enregistrement (Fiche ou Article) définit la structure de base des informations stockées dans un fichier.

Le langage Pascal traite un fichier de manière dynamique, c'est à dire que le programme peut faire varier la longueur du fichier. Le fichier possède un identificateur défini dans la partie de déclaration des variables.

Le type fichier est déclaré par les mots **FILE OF**, suivis du type des composants (de type quelconque, sauf fichier). Les mots FILE OF peuvent apparaître au niveau des déclarations de TYPE ou directement pour définir une variable dans la section VAR.

Syntaxe Pascal et Diagramme syntaxique

```
<type fichier> ::=
        FILE OF  <type des composants>
```



Exemples :

```
TYPE
  articles = FILE OF RECORD
          reference : INTEGER;
          libelle : packed ARRAY [1..10] of char;
          prixht : REAL
            END;
  piece = RECORD
          atelier : CHAR;
          refpiece : INTEGER;
          nom : packed ARRAY [1..15] of char;
          stock : REAL
          END;

VAR
  ficproduit : articles;
  ficpiece :   FILE OF piece;
  ficfacture : FILE OF RECORD
          objet : packed ARRAY [1..12] of char;
          paye : BOOLEAN
            END;
```

Dans ces déclarations, nous donnons volontairement aux variables fichiers des identificateurs commençant par les lettres *fic*.

- *piece* est un type enregistrement, tandis que *ficpiece* est une variable fichier constitué de composants de type *piece*,
- *articles* est un type fichier et *ficproduit* est un fichier de type *articles*,
- *ficfacture* est une variable fichier dont les composants sont directement définis au niveau de la déclaration de la variable.

Exemple de Notation algorithmique

```
Enregistrement : facture
    chaîne 12 caractères : objet
    logique : payé
Fin Enregistrement

Fichier de facture : ficfacture
```


Une variable de type fichier est structurée par la définition de ses composants, qui peuvent être de type standard ou d'un type défini par le programmateur

Nous avons donné des exemples de fichiers composés d'enregistrements ; les fichiers peuvent aussi être définis avec des composants simples entiers, réels, caractères ou booléens.

```
VAR
    ficentier : FILE OF INTEGER;
    ficreel :   FILE OF REAL;
    ficbool :   FILE OF BOOLEAN;
    ficcarac :  FILE OF CHAR;
```

Les fichiers de caractères sont très utilisés parcequ'ils contiennent en réalité des textes. Le langage Pascal prévoit à cet effet un fichier de type standard : le type TEXT, défini plus loin.

ACCES AU FICHIER

Un fichier séquentiel est composé d'une suite d'enregistrements disposés les uns à la suite des autres, telle que la lecture d'un enregistrement n'intervient qu'après la lecture des enregistrements précédents. Le support du fichier se comporte comme une bande magnétique devant se dérouler devant une tête de lecture.

Le Pascal standard ne traite que les fichiers séquentiels.

Certaines versions du langage Pascal (UCSD, Turbo-Pascal, MS-Pascal) permettent d'atteindre directement un enregistrement du fichier, quelque soit sa position, en utilisant une instruction de recherche qui contient le numéro ou la clé de l'enregistrement recherché (On parle de fichiers à accès direct et de fichiers indexés).

La liaison, entre la variable fichier déclarée dans le programme, et le fichier stocké sur un support externe (bande magnétique, disque, disquette), n'est pas définie par le Pascal standard et ne fait pas partie de la Norme NF Z65-300

Cette liaison est définie par la version du langage Pascal utilisé

Ainsi, Turbo-Pascal et MS-Pascal utilisent l'instruction ASSIGN pour associer un identificateur de fichier avec son nom réel (exemple pag 92)

FENETRE DE FICHIER

Une déclaration de variable de type Fichier engendre automatiquement une variable appelée FENETRE du fichier, de même type que le composant de base du fichier.
La FENETRE du fichier, ou variable TAMPON, contient le composant correspondant à la position courante du fichier.

Pour le fichier *ficentier*, la variable fenêtre est notée **ficentier ^**, et désigne l'entier accessible à un instant donné dans le fichier.
Pour le fichier *ficfacture*, la variable fenêtre est notée **ficfacture ^** , et désigne l'enregistrement accessible. Les champs du composant de base (voir déclaration de *ficfacture*) sont repérés par :

ficfacture ^.objet et
ficfacture ^.paye

Une variable fenêtre, comme toute variable, peut faire l'objet d'affectation, de lecture et d'écriture ainsi que des opérations spécifiques à son type.

Exemples :

```
READLN(ficentier ^ );
WRITELN('Valeur lue = ', ficentier ^ );
ficfacture ^ .objet := 'Lot 12 colis';
ficfacture ^ .paye := FALSE;
WRITELN('ficfacture ^ .objet);
```

PROCEDURES ET FONCTIONS DE MANIPULATION

1°- FONCTION FIN DE FICHIER EOF( )

Tout fichier Pascal est marqué d'un caractère de Fin de Fichier, indiquant la dernière position du fichier. **La fonction booléenne EOF( )** permet de tester si la fenêtre est positionnée ou non sur ce caractère terminal. EOF est l'abréviation de END OF FILE, qui signifie Fin de fichier.
La fonction EOF( ) admet pour paramètre l'identificateur du fichier à tester. Si la fonction retourne la valeur TRUE (VRAI), le contenu de la variable fenêtre n'est pas défini, et toute tentative de lecture provoquerait une erreur du programme.

La lecture d'un fichier commence donc toujours par un test de NON Fin de Fichier

Le test pour les fichiers FICPIECE et ficfacture peuvent s'écrire sous les formes suivantes :

IF **NOT EOF(ficpiece)** THEN ...
ou WHILE **NOT EOF(ficfacture)** DO ...

La structure REPEAT est bien sûr à bannir puisque le test n'est réalisé qu'après l'action pouvant provoquer une erreur de lecture dans un fichier vide.

2°- PROCEDURES DE FICHIER REWRITE( ) et RESET( )

La **procédure REWRITE( )**, notée REECRIT en Algorithmique, est utilisée pour **ouvrir un fichier en écriture.** Cette procédure admet pour paramètre l'identificateur du fichier sur lequel on souhaite travailler.
Un fichier ouvert dans le mode REWRITE crée un fichier vide : Si le fichier existe déjà, son contenu est perdu.

```
REWRITE(ficfacture);
```

Après l'instruction REWRITE, la fenêtre du fichier est positionnée sur le début d'un fichier vide. La procédure REWRITE affecte à la fonction EOF( ) la valeur VRAI. L'écriture est alors possible et on peut ajouter de nouveaux composants.

La **procédure RESET( )**, notée RELIT en Algorithmique, est utilisée pour **ouvrir un fichier en lecture.** Elle admet pour paramètre un identificateur de fichier.
Un fichier ouvert dans le mode RESET peut être lu, mais l'écriture est alors impossible.

```
RESET(ficpiece);
```

Après l'instruction RESET, la fenêtre du fichier est positionnée sur le premier composant et la variable fenêtre prend la valeur de ce composant (indéfini si le fichier est vide).
La procédure RESET affecte à la fonction EOF( ), la valeur VRAI si le fichier est vide et FAUX dans le cas contraire.

3°- PROCEDURES DE FENETRE PUT( ) et GET( )

L'écriture dans un fichier fait appel à la procédure **PUT**, notée MET en Algorithmique. La procédure PUT admet pour paramètre l'identificateur de fichier (*fic* dans notre exemple) dans lequel on écrit.

L'écriture se fait en 2 temps :
– Premier temps : Affectation à la variable fenêtre de la valeur du composant (ici *compos*) à écrire,
– Second temps : Ecriture de la variable fenêtre, en fin de fichier.

```
fic ^ := compos;
PUT(fic);
```

La lecture d'un fichier utilise la procédure GET, notée OBTIENT en Algorithmique. La procédure GET admet pour paramètre l'identificateur de fichier (*fic* dans notre exemple) dans lequel on lit.

La lecture se fait en 2 temps :
– Premier temps : Sauvegarde du contenu de la variable fenêtre dans une variable (ici *compos*), de même structure que le composant de base,
– Second temps : Déplacement de la fenêtre de fichier sur le composant suivant, et affectation à la variable fenêtre de la valeur de ce composant.

```
compos := fic ^;
GET(fic);
```

■ EXEMPLE NUMERIQUE SIMPLE

Saisir puis afficher une série de nombres entiers positifs, enregistrés dans un fichier (***fentier*** dans le programme et ***fic1*** en stockage physique).

```
PROGRAM exemplefic1 (INPUT,OUTPUT);
VAR                          (* Testé en MS-Pascal *)
  fentier : FILE OF INTEGER;
BEGIN                        (* Relation entre niveaux *)
  ASSIGN(fentier,'fic1');    (* logique et physique *)
  REWRITE(fentier);          (* Ouvert en Ecriture *)
  WRITE('Saisir un entier (<0 pour fin) ' );
  READLN(fentier ^);         (* Saisie de la fenêtre *)
  WHILE fentier ^ >= 0 DO
    BEGIN
      PUT(fentier);          (* Ecriture dans fichier *)
      WRITE('Saisir un entier (<0 pour fin) ' );
      READLN(fentier ^)
    END;
  RESET(fentier);            (* Ouvert en Lecture *)
  WHILE NOT EOF(fentier) DO
    BEGIN
      WRITELN(fentier ^);    (* Affichage écran*)
      GET(fentier)           (* Lecture *)
    END
END.
```


LES FICHIERS TEXT

1°- DEFINITION ET DECLARATION

Un fichier de composants de type caractère peut être prédeclaré de type **TEXT**. Dans ce cas, il est structuré en lignes. Chaque ligne, de longueur variable, est terminée par un caractère ou marque de Fin de Ligne.

La fonction booléenne EOLN( ) permet de tester si la fenêtre du fichier est positionnée sur la marque de Fin de Ligne. EOLN( ) est l'abréviation de END OF LINE signifiant Fin de Ligne.

2°- PROCEDURES EQUIVALENTES READ( ) et WRITE( )

Les lecture et écriture dans un fichier texte (soit *fictxt*), peuvent être réalisées, caractère par caractère (soit *car*), à partir des procédures standards READ, READLN, et WRITE, WRITELN.

READ(**fictxt, car**); équivaut à **car := fictxt ^ ;**
GET(fictxt);
La fenêtre se positionne sur le caractère suivant.

READLN(**fictxt**); termine la lecture de la ligne courante et positionne la fenêtre sur le 1er caractère de la ligne suivante.

WRITE(**fictxt, car**); équivaut à **fictxt ^ := car;**
PUT(fictxt);

WRITELN(**fictxt**); termine l'écriture sur la ligne courante du fichier texte *fictxt*, l'écriture du caractère suivant se fera sur une nouvelle ligne.

Par extension du Pascal standard, **la plupart des versions de Pascal autorisent les procédures READ et WRITE, pour tout type de fichier.**

APPLICATION V-8

Créer un fichier texte et saisir les 3 lignes :
'1. il faudra ajouter une'
'2. quatrième ligne en utilisant'
'3. une technique adaptée de'
Créer un 2ème fichier texte et saisir la ligne :
'4 recopie dans un troisième fichier'
Créer un 3ème fichier et générer son contenu de 4 lignes par recopie du fichier 1 et du fichier 2.

b) Programme Pascal

```
PROGRAM copiefichier (INPUT,OUTPUT);
VAR car : CHAR;              (* Testé en MS-Pascal *)
    f3ligne, f1ligne, f4ligne : TEXT;

PROCEDURE saisi(VAR fic :TEXT; n : integer);
VAR i:INTEGER;
BEGIN
  WRITELN('[Entree] pour finir 1 ligne ');
  FOR i := 1 TO n DO
    BEGIN
      REPEAT
        READ(car);
        WRITE(fic,car)
      UNTIL EOLN;
      WRITELN(fic)
    END;
END;

PROCEDURE copien3(VAR fic :TEXT);
BEGIN
  RESET(fic);
  WHILE NOT EOF(fic) DO
    BEGIN
      WHILE NOT EOLN(fic) DO
        BEGIN
          READ(fic,car);
          WRITE(f4ligne,car)
        END;
      READLN(fic);
      WRITELN(f4ligne)
    END;
END;

BEGIN                        (* Programme principal *)
  ASSIGN(f3ligne,'fic1');
  ASSIGN(f1ligne,'fic2');
  ASSIGN(f4ligne,'fic3');
  WRITELN('Saisir le texte du 1er fichier');
  REWRITE(f3ligne); saisi(f3ligne,3);
  WRITELN('Saisir le texte du 2eme fichier');
  REWRITE(f1ligne); saisi(f1ligne,1);
  REWRITE(f4ligne);
  copien3(f3ligne);
  copien3(f1ligne)
END.
```

Remarque 1 : ASSIGN est une procédure spécifiant au programme que tel fichier logique utilisé dans le programme correspond à tel fichier physique sur disque.

Remarque 2 : Les procédures saisi (1 paramètre par variable de type fichier TEXT et 1 paramètre par valeur de type entier) et copien3 (1 paramètre fichier par variable) sont appelées chacune 2 fois.

A – MOTS RESERVES
IDENTIFICATEURS PREDEFINIS

MOTS RESERVES		
AND	FUNCTION	PROGRAM
ARRAY	GOTO	RECORD
BEGIN	IF	REPEAT
CASE	IN	SET
CONST	LABEL	THEN
DIV	MOD	TO
DO	NIL	TYPE
DOWNTO	NOT	UNTIL
ELSE	OF	VAR
END	OR	WHILE
FILE	PACKED	WITH
FOR	PROCEDURE	

IDENTIFICATEURS PREDEFINIS		
CONSTANTES		
FALSE	TRUE	MAXINT
TYPES		
BOOLEAN	INTEGER	TEXT
CHAR	REAL	
VARIABLES		
INPUT	OUTPUT	
FONCTIONS		
ABS	EXP	SIN
ARCTAN	LN	SQR
CHR	ODD	SQRT
COS	ORD	SUCC
EOF	PRED	TRUNC
EOLN	ROUND	
PROCEDURES		
DISPOSE	PUT	UNPACK
GET	READ	WRITE
NEW	READLN	WRITELN
PACK	RESET	
PAGE	REWRITE	

B – INDEX

Adresse 6, 57, 59, 86, 88
Affectation 20, 25, 61, 79, 82
Algorithmique 8, 9
Alphabet 5, 6, 11, 25
Alternative 29, 31, 33, 34, 36
AND 27, 28, 33, 34, 75, 81
Apostrophe 21
Appartenance 79
Appel 50, 51, 53, 57, 59, 65
Argument 24, 61
ARRAY 67, 68, 71
Article 82
Assembleur 6
Assignation 20
Avec 83, 85
BEGIN 14, 16, 22, 29, 30, 32
Binaire 5, 6
Bloc 12, 13, 50, 51, 61
BNF 13
BOOLEAN 15, 17, 19, 30
Boucle 42, 44, 53, 69, 77, 88
Caractère 11, 14, 16, 19
CASE 35, 36, 37, 41
Chaîne 12, 14, 16, 21, 72, 83
Champ 82, 83, 84, 85
CHAR 15, 16, 17, 19
Chiffre 11, 15, 22
CHR 19
Commentaire 21, 24, 56
Compactage 72, 74, 83
Compilateur 7
Condition 29, 30
Condition Multiple 35, 36
CONST 16, 18, 21, 25
Constante 15, 16, 17, 25
Corps 13, 25
Début action 14, 15, 16, 17
Début algorithme 14, 15, 17
Déclaration 13, 14, 16, 18, 51
Diagramme 13
Différence 79
Dimension 66, 68, 72, 76, 77
DISPOSE 87
DIV 23, 26, 64
DO 39, 40, 41, 43, 44, 60
Dynamique 86, 87
Ecrire 21, 22
Effectif (Paramètre) 57, 59, 62
Elément 38, 42, 66, 79, 86
En-tête 13, 16, 51, 61
END 14, 16, 22, 28, 31, 82
Enregistrement 82, 84, 87, 90
Ensemble 79, 80
Entier 15, 16, 17
Entrée Sortie 9, 20
Enumération 19
EOF 91, 92
EOLN 73, 93
Espace 11, 14, 16, 22, 73
ET 27, 28, 34
Etiquette 15, 35
Expression 23, 24, 27, 31, 35
Faire 38, 39, 42, 43
FALSE (FAUX) 15, 17, 19, 27
Fichier 14, 20, 90
File 38, 42, 45
FILE OF 90, 92
Fin action 10, 14
Fin algorithme 10, 15
Fonction 19, 24, 61, 64
FOR 39, 40, 62, 69
Format 22
Formel (Paramètre) 57,59, 61
FUNCTION 61, 63
GET 92, 93
Globale (Variable) 54, 56
GOTO 15
Hiérarchie 24, 49
Identificateur 14, 16, 25, 51
IF THEN ELSE 30, 32, 34, 36
Imbrication 56, 83
IN 79, 81
Incrémentation 20, 23, 39
Indentation 30
Indice 38, 39, 66, 68, 72, 76
Initialisation 25, 41, 68
INPUT 14, 16
Instruction composée 29, 31
Instruction simple 20, 30
INTEGER 15, 17, 19
Interpréteur 7
Intersection 79
Intervalle 18, 19, 67
Itération 41, 42, 45
LABEL 15
Langage 5, 6, 7, 8, 9, 11, 16
Lettre 11, 14, 15
Lire 20, 21, 24, 26
Liste chaînée 88, 89
Locale (Variable) 54, 56, 58
Logique 15, 17, 27, 29
MAXINT 16, 40
Mémoire 5, 6, 65, 72, 86, 87
Message 30, 34, 56, 74
MOD 23, 26, 30, 64
Mots clés, réservés 3, 9, 10
NEW 87, 88, 89
NIL 88, 89
Norme 9, 12
NOT (NON) 27, 28, 31, 54, 60
Objet 7, 15, 17, 25
ODD 27
Opérande 23
Opérateur 19, 23, 24, 27, 79
OR (OU) 27, 28, 34, 60
ORD 19
Ordinateur 5, 7
Organigramme 9
OUTPUT 14, 16
PACKED 72, 73, 77
Paramètre valeur 57, 58, 59
Paramètre variable 57, 58, 59
Point décimal 21, 22
Point virgule 22, 32, 45
Pointeur 86, 87, 88
Portée 54, 55, 56
Pour 38, 39, 40
PRED 19
Priorité 24, 27
PROCEDURE 51, 52, 57, 58
PROGRAM 14, 16, 18
PUT 92, 93
READ 20, 21, 22, 93
REAL (Réel) 15, 16, 17, 22
RECORD 82, 83, 85, 87, 90
Récursivité 65
REPEAT (Répéter) 45, 47, 53
RESET 92, 93
REWRITE 92, 93
Scalaire 18, 19, 79
Sélecteur 35, 36, 83
Séquentiel 91
SET OF 79, 80, 81
Si Alors Sinon 29, 30, 31, 33
Source (Programme) 7
Sous-programme 50, 53, 57
STRING 72, 73
SUCC 19
Syntaxe 13
Tableau 66, 67, 76, 77
Test 29, 33, 35, 36
TEXT 93
Trace 10, 31, 69
Traducteur 7
TRUE 15, 17, 19, 27, 29, 35
Type 15, 16, 18
Union 79
VAR 17, 18, 22
Variable 10, 17, 18, 19
Variante 83, 84
Vecteur 66, 67, 68
VRAI 15, 17, 19, 20, 27
WHILE (Tant que) 42, 43, 79
WITH 83, 85
WRITE 21, 22, 24, 93